SYNERGISTIC ENGINEERING ENVIRONMENT BUILD II

DEVELOPER GUIDE

Prepared by
Doug Murphy

Analytical Mechanics Associates
Hampton, VA

Revision E
AMA Report No.03-31
NASA Prime Contract No.L71131D
August 2003

TABLE OF CONTENTS

LIST OF FIGURES

1 INTRODUCTION

1.1 Identification of Document

This is the Developer Guide for the Synergistic Engineering Environment (SEE) Build II software. This guide and the SEE software was prepared by Analytical Mechanics Associates, Hampton VA. For inquiries please contact Doug Murphy at Doug@ama-inc.com.

1.2 Scope of Document

This revision of the Developer Guide is applicable to release 1.0 of the SEE Build II software, released August 2003.

1.3 Purpose and Objectives of Document

This document is intended to provide background on the SEE Build II software to aid future developers in planning upgrades, adding new features and integrating new analysis tools. As of this release of the SEE, information about using this software for the Comet and Asteroid Protection System (CAPS) project is available in a separate report. Section 2 contains full cites for applicable documents and resources, including the SEE User Guide and the SEE CAPS Module Guide.

1.4 Document Status and Schedule

This is revision E of the Developer Guide. Revisions are planned for each future software release in which substantial architecture changes or additions are made.

Revision	Date	Change Log
-	March 2002	Initial Release
A	July 2002	Added discussion of *Solar System Object* inheritance hierarchy, scene graph building and classes for interfacing with dynamics models. Included references to the SEE CAPS Module Guide.
B	December 2002	Added description of solar system initialization, mission wizard, SEE object libraries, collision detection and image capture.
C	February 2003	Added description of plotting, analysis tools, Bookmarks. Plus misc. revisions.
D	April 2003	Added descriptions of movie capture, macros, "New Craft" wizard, jet plume visualization.
E	August 2003	Simulation time description updated. Qt Assistant description added. Dynamics model section revised. DDEX file format diagram revised.

1.5 Document Organization

The following fonts are used to highlight special terms:

`courier new`	Program names, C++ class names, code fragments or variable names.
italic	Terms with specific connotation in the SEE context.

2 RELATED DOCUMENTATION

2.1 Applicable Documents

These documents are referenced herein or are directly applicable to the SEE architecture:

1. *Synergistic Engineering Environment Build II, Guide to the Comet and Asteroid Protection System (CAPS) Module,* Analytical Mechanics Associates Inc., Hampton VA, July 2002.

2. *Synergistic Engineering Environment Build II Users Guide*, Analytical Mechanics Associates Inc., Hampton VA, August 2003.

3. *On-Orbit Assembly, Modeling, and Mass Properties Data Book, Volume I,* International Space Station Program, DAC8, Revision Sequence E, NASA Johnson Space Center, August 1999.

4. *Method for Propagating the Orbital Elements of the Solar System for the Synergistic Engineering Environment*, AMA Report No. 01-28, Analytical Mechanics Associates Inc., Hampton VA, 2001.

5. *The Astronomical Almanac For the Year 2003,* Nautical Almanac Office, United States Naval Observatory, U.S. Government Printing Office, Washington, D.C., 2002.

6. Bate, R., Mueller, D. and White, J., *Fundamentals of Astrodynamics*, Dover Publications, New York, 1971.

7. Battin, R.H, *An Introduction to the Mathematics and Methods of Astrodynamics, Revised Edition*, American Institute of Aeronautics and Astronautics, Inc., Virginia, 1999, pp. 194.

2.2 Information Documents

The OpenGL Programming Guide, Third Edition, Mason Woo, Jackie Neider, Tom Davis, Dave Shreiner, OpenGL Architecture Review Board, Addison-Wesley, Reading, MA 1999.

International Space Station Synergistic Engineering Environment Version Beta 0.60 Users Guide, Analytical Mechanics Associates Inc., Hampton VA, December 2001.

Architectural Design Description of the International Space Station Synergistic Engineering Environment, Analytical Mechanics Associates Inc., Hampton VA, December 2001.

2.3 On-Line Resources

These on-line resources are used in the development and configuration management of the SEE project:

http://draco.ama-inc.com/iss/
 The developer home page of the SEE for Build I and II. Contains links to documentation, images and past analyses.

http://adapt-www.larc.nasa.gov:23456/SEE/
 The SEE BookBuilder web site. Contains requirements documents, schedules and design notes.

http://centauri.larc.nasa.gov/see
 The SEE public website. Provides a summary of the current and planned capabilities of the SEE. Links to the see developer home page where registered users can log-in and submit bug reports and feature requests.

ftp://griffin.larc.nasa.gov/usr/people/medusa3/shield/cvsroot/seebuild2
 The CVS repository for the SEE source code.

3 MOTIVATION AND GOALS

3.1 Background

Build I of the SEE, known previously as the ISS SEE, was originally commissioned to demonstrate prospective features of a new engineering tool for visualizing the combined results of analysis programs used for work on the International Space Station. The final release of the ISS SEE Build I was completed in December 2001. Build II is a follow-on to that project, and aims to continue and further the capabilities of Build I with the focus expanded to include the modeling and visualization of solar system exploration scenarios in addition to operations of the ISS.

Build I of the SEE made use of the MuSE 3D commercial application programming interface (API) for management of the graphical scene. Support for the MuSE software was discontinued by the vendor at the end of 2001. By that time it was also becoming clear that the SEE Build I software architecture had reached a point where the addition of new requirements would require significant changes to the core code. In order to continue to support the continued development of the SEE, a complete re-write of the code was undertaken for Build II.

3.2 SEE Build II

The goal of the SEE Build II project is to design a space mission simulator with emphasis on providing an interactive three-dimensional virtual environment in which to view the results. Experience with the Build I project showed that the ability to examine the geometry of the vehicles and solar system bodies moving in the scene, as prescribed by the results of analysis tools, was beneficial both in understanding the scenario and communicating the results to others[1].

As with Build I, a major design goal of Build II is to include the capability to allow the user to combine a variety of data sets into a single environment. In Build I this data included spacecraft positions and attitude, joint articulation data, thruster jet firing histories, robot arm movements and other properties of spacecrafts. The planets and moons of the solar system were simulated exclusively by an engine internal to the SEE program. By the end of the Build I development, work on features for importing planet and moon position and orientation data was just beginning. The design of Build II will emphasize the capability to select internal or external data sources for *any* object in the scene[2].

Where possible and efficient, the SEE will aim to provide methods that interface directly with external analysis tools. This type of interface would allow the user, for example, to identify a spacecraft currently being viewed in the SEE environment for a new dynamics analysis. The SEE program would then export the properties of the selected spacecraft in a format compatible with the target analysis tool, run the analysis, and import the results.

3.3 Requirements

The specific requirements for features desired in the Build II SEE were based on those for Build I. Features that were seldom used on Build I have been de-emphasized or scheduled for late development on Build II, so that more commonly used features will get priority. The requirements for new or refined capabilities have come from team leaders and team members in the Revolutionary Aerospace Systems Concepts (RASC) Activity at Langley Research Center (LaRC), and also from members of the International Space Station (ISS) Vehicle Integrated Performance and Resources (VIPER) team at Johnson Space Center. RASC projects that have or intend to use the SEE include the Hybrid Propellant Module and the Comet and Asteroid Protection System (CAPS). The *SEE CAPS Module Guide* contains detailed requirements for CAPS features of the SEE. Requirements from the VIPER team focus on the systems engineering aspects of the ISS.

The top-level requirements for the system include the capability to run on Windows PCs, as well as the Linux and IRIX operating systems. The software will serve primarily as a space mission visualization tool but will be extensible to allow developers to integrate analysis algorithms as required by the users.

4 ARCHITECTURAL DESIGN DESCRIPTION

4.1 Tools

The SEE project is comprised of several related tools. The SEE executable is the primary application. Support applications include the Dynamics Data Extraction Program (DDEX) and the Articulated Rigid Body Control Dynamics Program (ARCD). DDEX is a tool used for converting CAD databases to SEE compatible format, and consists of a C++ program and CAD scripts. ARCD is a Fortran program developed by NASA and is used for evaluating the dynamics of a spacecraft in Earth orbit. Most of the functionality of the SEE program is independent of these support tools. They are not required to be part of an SEE distribution to a particular customer.

The SEE main program source code is object-oriented and written in C++. Several third party libraries are utilized. The 3D graphics API is *Gizmo 3D*. The window library is *QT*, including the *Qwt* extension for plotting features. Collision detection utilizes the *PQP* libraries. Gizmo and QT are briefly described below. PQP is discussed further in section 4.13.

4.2 Gizmo 3D

The decision to search for an existing high-level graphics API to replace MuSE, rather than to develop a custom library, was motivated by the development schedule. It was determined that most of the graphics requirements of the SEE were likely to be found in an existing API, and that the time invested in learning to use that library would yield a better return than developing and testing a custom solution. The desired requirements for the graphics API included Windows, IRIX, and Linux platform compatibility, fast rendering algorithms capable of supporting up to 500,000 polygons at interactive rates, availability of double precision support for transformation matrices, and free or affordable run-time licenses. Support for double precision appeared to be the limiting factor for most of the prospective choices.
Gizmo 3D from Tooltech Software met the list of requirements and passed preliminary performance evaluations, including a small program designed to stress the frame rate by using high polygon-count models. Gizmo is available for Windows, IRIX and Linux, and contains built-in routines for loading popular tessellated model formats such as Discrete's *3D-Studio* and SGI's *Performer*. The Gizmo API is built upon OpenGL and adds functions for building a scene graph using node objects. The scene graph aspects of the library appear to be similar to the functions provided by *Performer* and *Inventor*. A drawback of the Gizmo API as of this writing is a lack of documentation.

4.3 QT

For the same reason that an existing 3D API was sought, it was decided to use an available software package for creating cross-platform graphical user interfaces. *QT* from Trolltech software was selected to replace Tcl/TK – the library used for SEE Build I. The QT libraries are written in C++, thus the QT function calls can be integrated directly with the SEE source code. This advantage allows the GUI signals to be communicated to the SEE functions without the need for additional code to handle inter-process communications[3]. QT has a significant user-base, is available for the three platforms of interest, and comes with a visual GUI building environment for automatic code generation. While Gizmo itself provides some limited interaction tools to allow the developer to detect keyboard and mouse events, it contains no features for creating the familiar graphical "widgets" (e.g. scroll bars and push buttons). The authors of Gizmo have recently identified QT as their primary API for the development of GUI interaction with the Gizmo system.

The *Qwt (QT Widgets for Technical Applications)* library is an open source project that adds plotting routines and other functions to QT. The plotting capabilities of the SEE make use of this library. The Qwt project web page is at http://sourceforge.net/projects/qwt.

4.4 Functional Decomposition

The routines and data files comprising the SEE can be thought of as belonging in one of the following function categories: user interface, visualization, simulation tools. One of the design goals for the project is to keep the source code divided into modules, based on their function, using these categories. This modularization is intended to aid in making the code extensible, reducing the file dependencies required for compilation, and allowing concurrent development. Appendix A shows the top-level directories for the files in the project with descriptive annotations. The source code for the main SEE executable is divided into subdirectories one level deep. With the exception of utilities such as the math libraries and the initialization codes in *main*, the sources in each of the subdirectories should support only one of the function categories.

The user interface category is comprised largely of QT objects (referred to as *widgets)*, which includes the application main window, the dialog boxes, and familiar graphical user interface widgets. The software architecture will support the custom configuration of the GUI, and allow the configuration to be saved separately for each user.

The Visualization category includes routines for building the graphical representations of the solar system model, spacecrafts, ground stations, asteroids, comets and other rendered items in the scene. For each of these items, an instantiated object of the appropriate C++ class is added to an SEE object hierarchy. Each SEE object in turn has a representation as a *gzNode* object in the Gizmo scene graph.

The SEE graphical objects are organized by type (Figure 2), with each type belonging to a manager class responsible for instantiating and updating the objects. The program execution proceeds through an event loop created by the *system manager*; a high level class responsible for the program flow of control. In each pass through the event loop the object managers update the position, orientation and other dynamic properties of all their child objects.

4.5 The Event Loop

The SEE program runs in an event loop as depicted in Figure 1. The initialization phase occurs at application startup. Part of the initialization phase includes instantiation of a QT *timer* object. The function of this timer object is to send out a signal that all user-request (e.g. keyboard and mouse inputs) have been received and processed by their respective signal handlers. When this signal is received by the system manager, the *update*, *draw*, and *respond* routines are called in order[4].

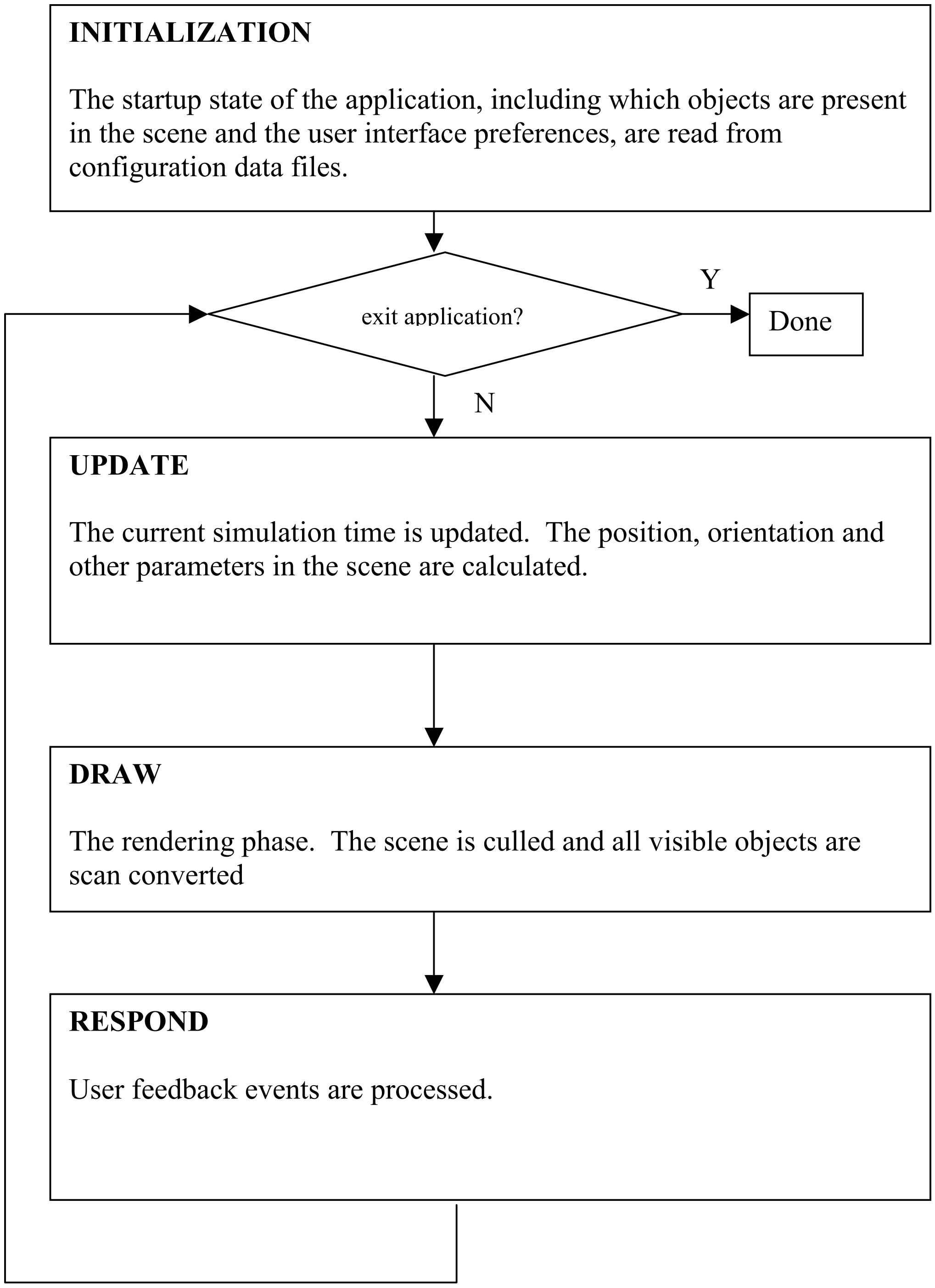

INITIALIZATION
The startup state of the application, including which objects are present in the scene and the user interface preferences, are read from configuration data files.
exit application?
Y
Done
N
UPDATE
The current simulation time is updated. The position, orientation and other parameters in the scene are calculated.
DRAW
The rendering phase. The scene is culled and all visible objects are scan converted
RESPOND
User feedback events are processed.

Figure 1: SEE Event Loop

4.6 SEE Objects

Figure 2 shows the arrangement of the top level classes used for SEE objects. Classes `Planet` and `Craft` are derived from class `SeeObject`. Class `SeeObject` has as a member `seeType` so that the child objects may be easily identified.

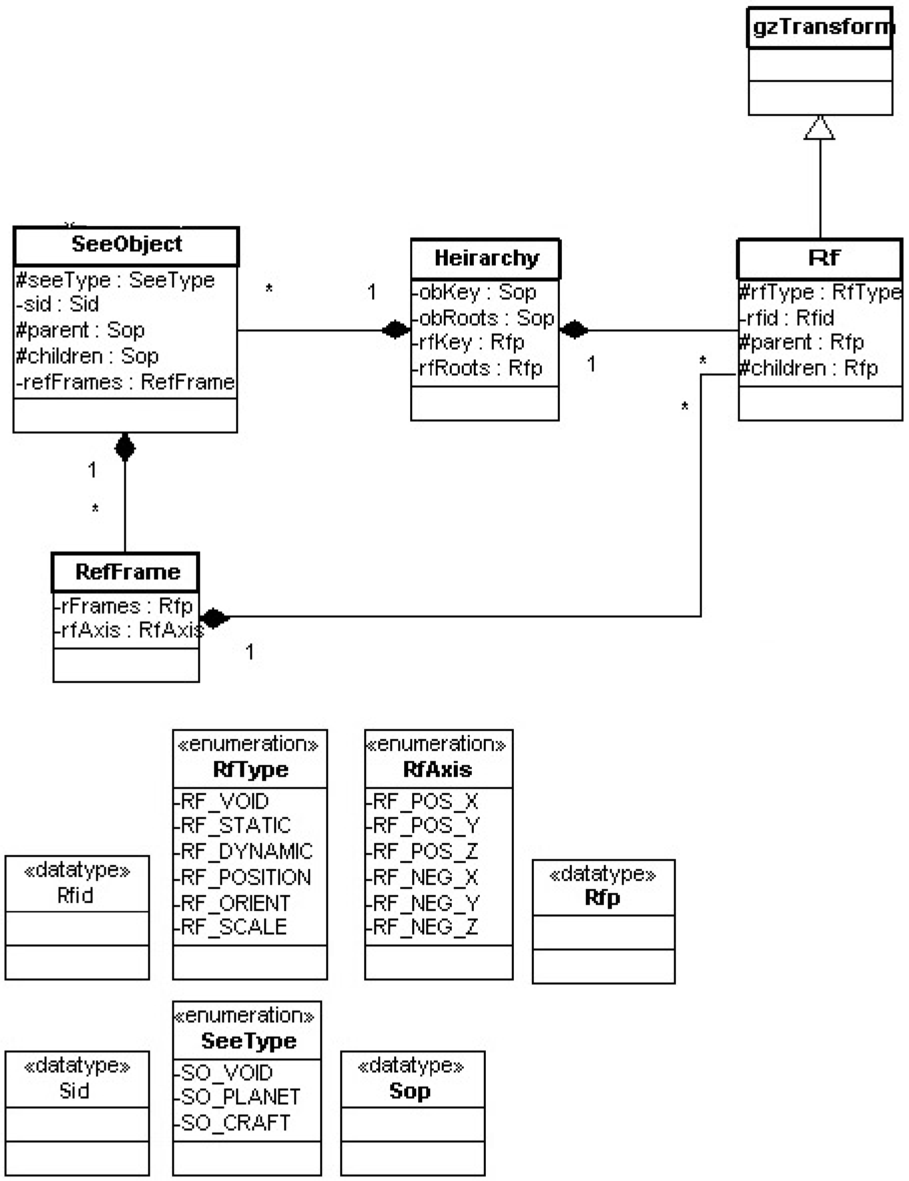


Figure 2: SEE Graphical Object and Reference Frame Hierarchy

Class `Hierarchy` is used to keep track of the parent-child relationships of the SEE objects and the reference frames to which they are attached. The distinction between the objects and their reference frames is important since the hierarchy may be different for each. The hierarchy of `SeeObjects` is designed to facilitate the way we are likely to think about the organization of the scene and is used to build the selection lists for the GUI. The hierarchy of `RefFrame` objects reflects the structure of the transformation matrices in the scene graph. To illustrate, if a user creates a new earth-orbiting satellite using a dynamics data set in a heliocentric reference frame, the `SeeObject` created for the satellite will have the `Planet` object “Earth” as a parent. However, since the dynamics data that describes the position and orientation of that satellite is relative to the Sun, the design reference frame of that satellite will have the inertial or Sun frame as a parent.

4.7 Hierarchies

There are three tree-structured hierarchies maintained within the SEE: the `SeeObject` hierarchy, the `Rf` hierarchy, and the scene graph. The `SeeObject` hierarchy maintains the inheritance structure of the all `SeeObjects`, primarily for the user interface purposes. For instance, since the Earth orbits the Sun, one would set the Earth as a child of the Sun. Likewise, a craft orbiting a planet would consider that planet as its parent (even if the data describing the orbit were heliocentric coordinates).

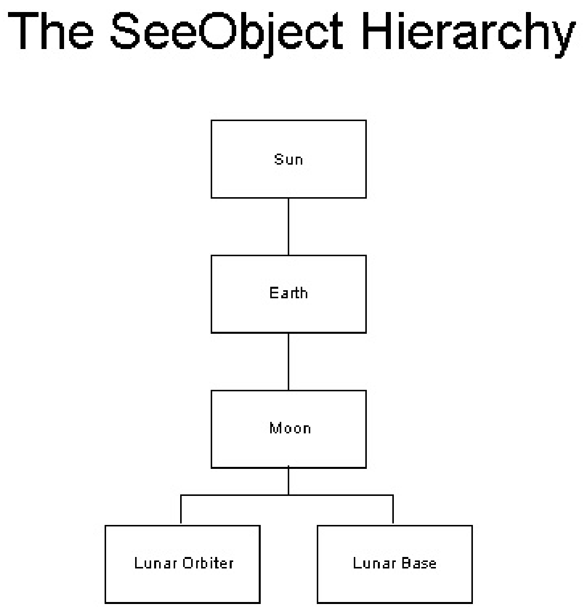


Figure 3: SEE Object Hierarchy

The scene graph is a structure of Gizmo3D objects, called *nodes*. A node could be anything from a positional transform to actual geometry. Gizmo3D traverses the scene graph a number of times during each rendering cycle. Except for camera position, all rendering information is stored within the scene graph.

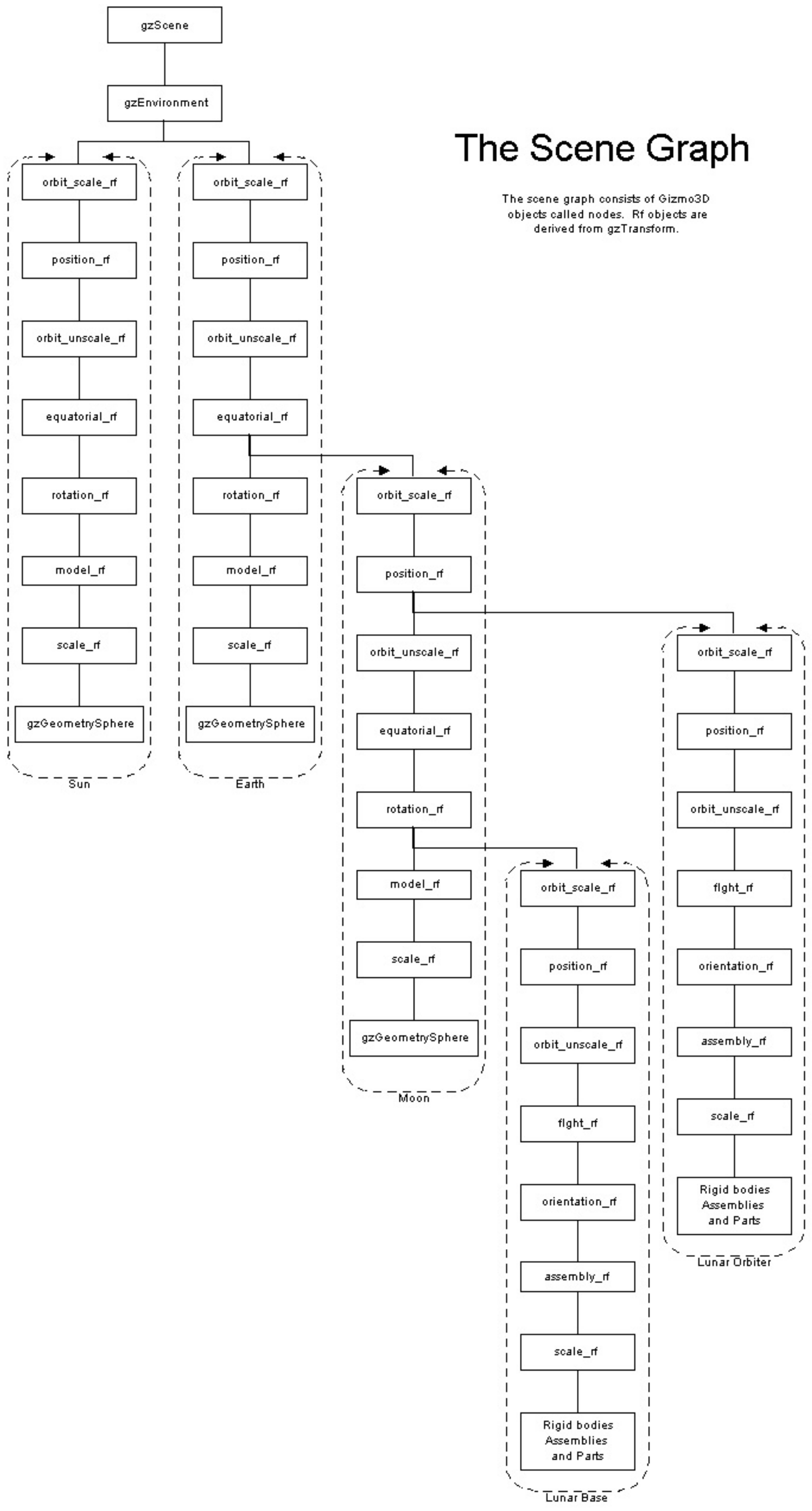


Figure 4: The Scene Graph

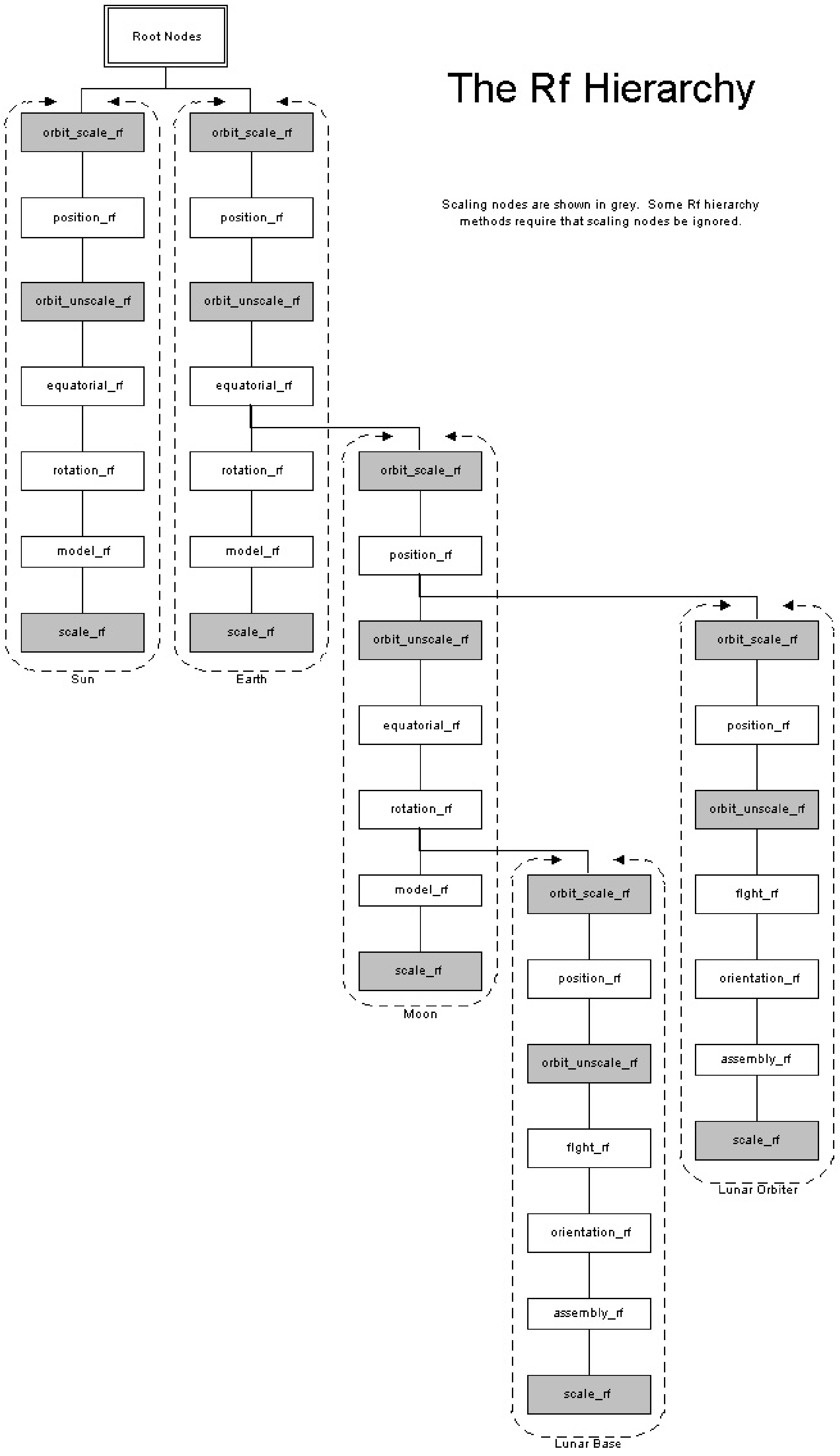


Figure 5: The Rf Hierarchy

The Rf hierarchy is a subtree of the scene graph, containing only the transforms. The *Rf* object is, in fact, derived from the *gzTransform* node. The Rf hierarchy is used to locate the positions of objects for either analytical or graphical purposes. For instance, a line of sight from the Earth to Mars requires the heliocentric positions of each. And tethering of cameras to objects requires their graphical positions (possibly different from the heliocentric position due to orbit scaling).

4.8 Solar System Objects

Real or physical objects are represented in the SEE by one of the subclasses of `SolarSystemObject` (Figure 6). These objects are distinguished by the fact that they must have mass properties, whereas other objects such as coordinate axes, vector arrows and camera icons may have a graphical representation in the scene but do not have mass. Class `SolarSystemObject` is abstract. Instantiated objects of the class will be of one of the inherited types: `SimpleObject`, `ArtificialObject`, or `NaturalObject`. Class `SimpleObject` is designed to represent either minor planets or crafts with no articulating parts. Crafts of this type will not be able to change their orientation to support a given flight mode. This class is intended to support the creation of a large number of objects such as an asteroid field or a large constellation of satellites, whose orbit data will likely be read from a data file or generated automatically rather than typed in via the GUI.
Class `ArtificialObject` encapsulates man-made objects such as satellites and space stations that are likely to have named flight modes, design reference frames, articulating parts and instruments. Currently the features to support flight modes and articulating rigid bodies are all implemented in the *Craft* subclass. The current version of `ArtificialObject` is actually a placeholder, which will be expanded in future SEE revisions as necessary. Models for craft subsystems such as power, thermal and ACS will interface with this class.
Class `NaturalObject` encapsulates features that support the simulation of planets, moons and minor planets. The orientation of these objects will be constructed using equatorial reference frames and spin axes or poles. Atmospheric and gravitational models will interface with this class.

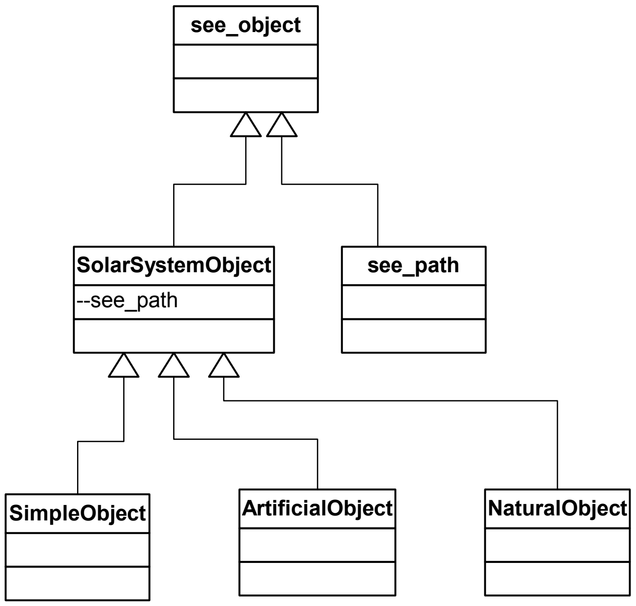


Figure 6: SolarSystem Class Inheritance

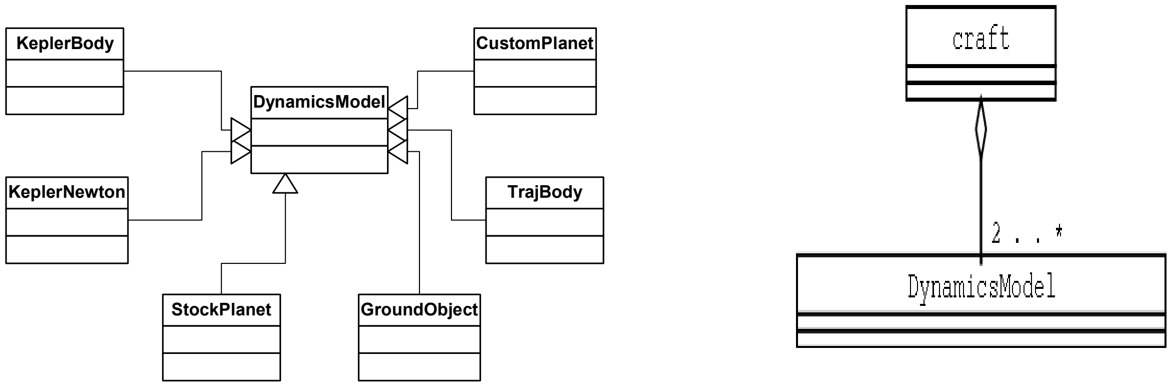


Figure 7: Classes Craft and Dynamics Model

4.9 Dynamics Models and Paths

All *real* objects being modeled in the SEE will have a path in space, and therefore must be connected to a system that generates time dependent position data. Algorithms that generate this data, and optionally orientation, velocity and acceleration data, are encapsulated in class `DynamicsModel`. To create a `SolarSystemObject` that requires a path through space, the SEE instantiates an appropriate `DynamicsModel` according to the parameters provided by the user (e.g. a model that calculates position and velocity according to Kepler's equations may take a series of orbital elements as initialization data). The `DynamicsModel` is used in conjunction with class `see_path` to ultimately generate the splines that will be used to render the object trajectory. Class `see_path` is used to provide a consistent interface between `SolarSystemObject`s and their paths. In practice, the must be `SolarSystemObject` must be of type `SimpleObject`, `Craft`, or `NaturalObject`, since assemblies and parts of crafts do not require independent paths.

Figure 7 shows the `DynamicsModel` classes and the relationship of class `craft` to `DynamicsModel`. Classes derived from `DynamicsModel` all provide, at a minimum, position versus time data relative to their parent SEE object in some form. Class `GroundObject` will return latitude, longitude, and altitude versus time. To resolve the heliocentric or planet-centric position of such an object, the SEE must also query the planet on which the object is placed for it's position and orientation data. Classes `KeplerBody` and `KeplerNewton` provide two methods for positioning an object according to Kepler's laws. These methods are described further in sections 4.9.1 and 4.9.2. Class `TrajBody` is designed to interface with a discrete set of position versus time data that might be stored in a file. Classes `StockPlanet` and `CustomPlanet` are specifically designed to support the placement and orientation of `NaturalObjects`. These classes are used to interface with "black-box" solar system simulations that have a library of planets and moons referenced by name. An example is the "Orrery", the SEE internal solar system simulator. Planets driven by the Orrery are instantiated with `StockPlanet` dynamics models. At update time these objects query the Orrery for the planet position and orientation by supplying only the planet (or moon) name and a simulation time. Class `CustomPlanet` is a placeholder designed to allow planets to be created using the same algorithms used in the Orrery but that have custom initialization data (e.g. an approximated Earth model with zero inclination and eccentricity).

In every craft there will be a dynamics model which generates the position and another dynamics model which generates the orientation. This orientation is generated by the `AttitudeManeuver` dynamics model. In the future, additional dynamics models could be implemented for a craft to control joint motions.

<u>Attitude Maneuvers Architecture</u>

The architecture of the `AttitudeManeuver` dynamics model supports multiple maneuvers. This relationship is shown in Figure 8. The `AttitudeManeuver` class has a `std::map< SEEdate, Maneuver*>` of `Maneuver` objects which are created for each of the craft's maneuvers. When the craft goes through an update, the current simulation time is sent to the `AttitudeManeuver` class. This simulation time is then checked against the starting and ending times of the craft's maneuvers. If any of the maneuvers are determine to be valid at this time, the calculation of the orientation matrix is computed in the `Maneuvers` class. There are three types of maneuvers: Absolute, Relative, and Constant Rotation. For Absolute maneuvers, the final orientation is specified with respected to the craft's flight mode. The `Maneuver` class interpolates the orientation of the craft as it updates during the specified time period. To accomplish this, the starting and ending orientation matrices are converted to quaternions and spherical linear interpolation is performed at each update. In the case of a Relative maneuver the only difference is that the starting matrix used is the current orientation of the craft at the start of the maneuver. The Relative maneuver is therefore just an offset of the craft's orientation. For the constant rotation case the craft is rotated about one of the three principal axes at a constant rate.

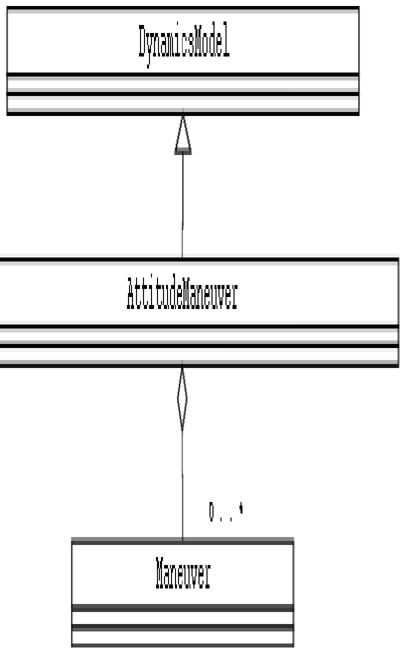


Figure 8: Maneuver classes

4.9.1 Dynamics Model Class: KeplerBody

The dynamics model class `KeplerBody` is used to implement the two-body model for positioning an object in an elliptical or circular orbit about a primary body. It accepts the classical orbit elements of an object (a, e, i, ω, Ω, M) and returns the position and velocity of that object in the same reference frame to which the orbital elements were referenced.

<u>Limitations:</u>
When calculating the true anomaly of the spacecraft for the current time from the value of mean anomaly, this routine uses an approximation [Ref. 5, pp. E4] given by:

$$\nu = M + (2e - e^3/4)\sin(M) + (5e^2/4)\sin(2M) + (13e^3/12)\sin 3M \ \ [+\ldots]$$

where ν is the true anomaly, *M* is the mean anomaly, and *e* is the orbit eccentricity. This approximation becomes less and less accurate as the eccentricity increases from a value of zero. It is very accurate for near circular orbits, and the error is zero for circular orbits. The error is not cumulative over multiple orbits, it is rectified each ½ orbit. The maximum error occurs somewhere between a true anomaly of 70 and 120 degrees and then again between 250 and 300 degrees, but the approximation yields zero error for values of M=ν=0 and M=ν=180.

Two error values were calculated for illustration. The maximum magnitude of the error in true anomaly (difference between actual and calculated), and the resulting maximum positional error of the object as a percent of the orbit semi-major axis. These error magnitudes are illustrated in Figure 9.

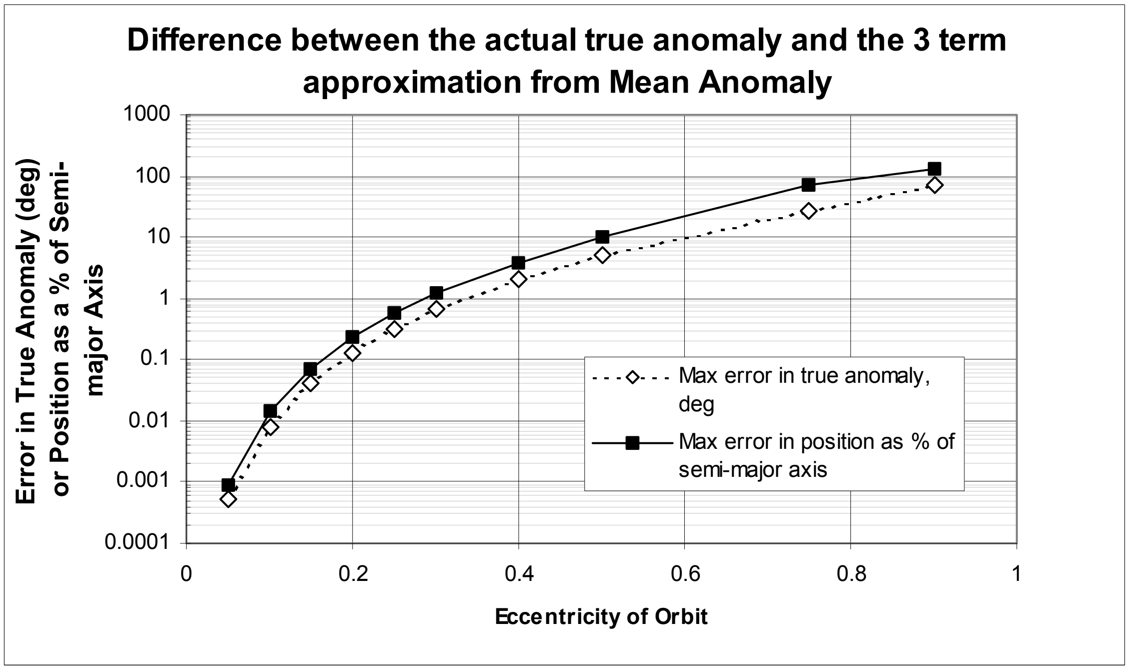


Figure 9: Maximum Error introduced into object position by Keplerbody approximation.

<u>Usage:</u>

`Keplerbody` provides a very rapid and computationally non-intensive way to obtain the position and velocity of an object given the orbital elements. It should only be used for objects with low orbit eccentricity (probably < 0.15, depending on allowable error) where speed of calculation is of primary concern, and absolute accuracy is secondary.

4.9.2 Dynamics Model Class: KeplerNewton

The `Keplernewton` dynamics routine in the SEE is to be used for elliptical and circular orbits about a primary body. It accepts the classical orbit elements of an object (a, e, i, ω, Ω, M) and returns the position and velocity of that object in the same reference frame to which the orbital elements were referenced.

The solution technique is one in which the true anomaly of the object is calculated from an iterative solution to Kepler's Equation:

$$M = E - e\sin(E)$$

where M is the mean anomaly, E is the eccentric anomaly, and e is the eccentricity of the orbit. The Newton iteration scheme used for solving for E given an initial guess at M is well documented [Ref. 6, pp. 221], and entails making a guess for E, calculating M, and then revising the guess for E based upon the result until convergence is achieved between the known value of M and that from the equation above.

At the time of model review, the initial guess for the first value of E in the iteration technique was taken from Battin [Ref. 7, pp. 194]:

$$E = M + \frac{e\sin(M)}{1 - \sin(M + e) + \sin(M)}$$

where M and E are measured in radians. This approximation provides a very good estimate for orbits with low eccentricity, or for all orbits where M is less than a value of π. It was noticed, however that when M was between π and 2π for orbits with high eccentricity the approximation became very poor as illustrated in Figure 10.

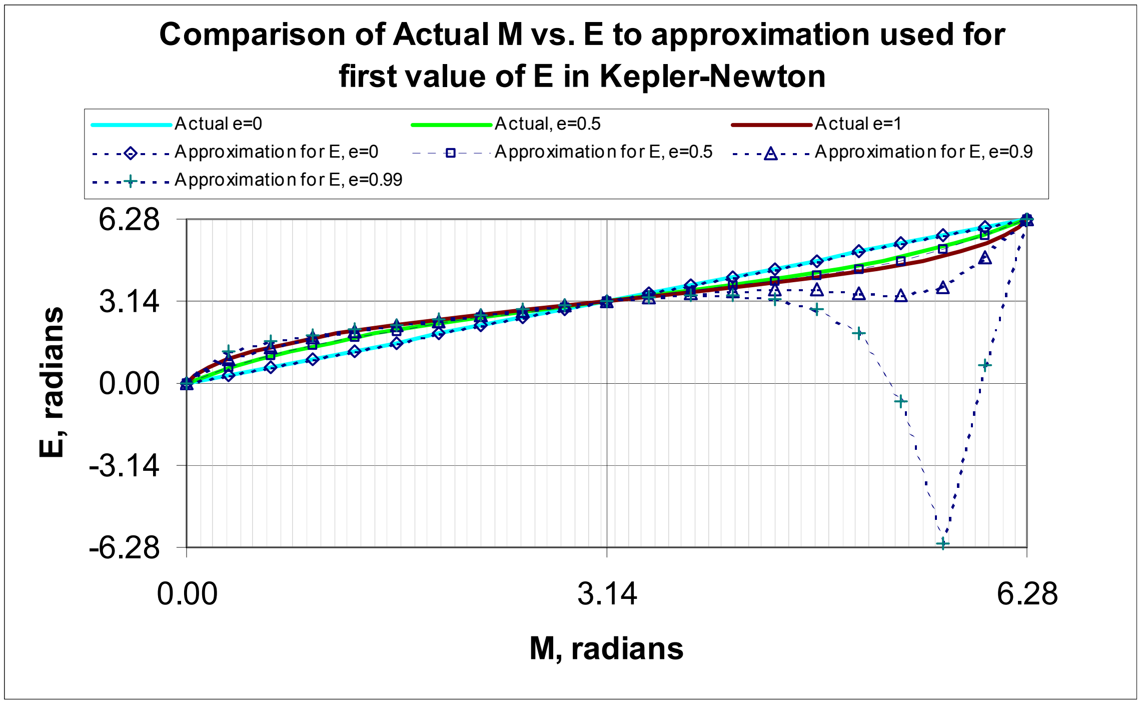


Figure 10. The real relationship between E and M as compared to Battin's approximation for initial guess.

This poor approximation actually led to a rapid increase in the number of iterations required for convergence on a value of E when M was near 5.3 radians. Although the typical Keplernewton solution should take 5-30 iterations to converge, for eccentricities above 0.9 the number of iterations escalated to the tens of thousands.

The Newton iteration technique is known to converge for all values of eccentricity (e) for an initial guess of E=π, but convergence can be slower than for an initial guess closer to the real value of E. The number of iteration steps required to converge for each initial guess approach was investigated. A Matlab script was created that performed the convergence iteration for both initial guesses at each value of E from 0 to 2π, and for e from 0 to 1. For each value of e, the maximum number of iterations required for any single convergence was extracted, and is shown in Figure 11. The figure clearly illustrates that for high eccentricity orbits (e > 0.9), that convergence is much more quickly performed by using π as an initial guess.

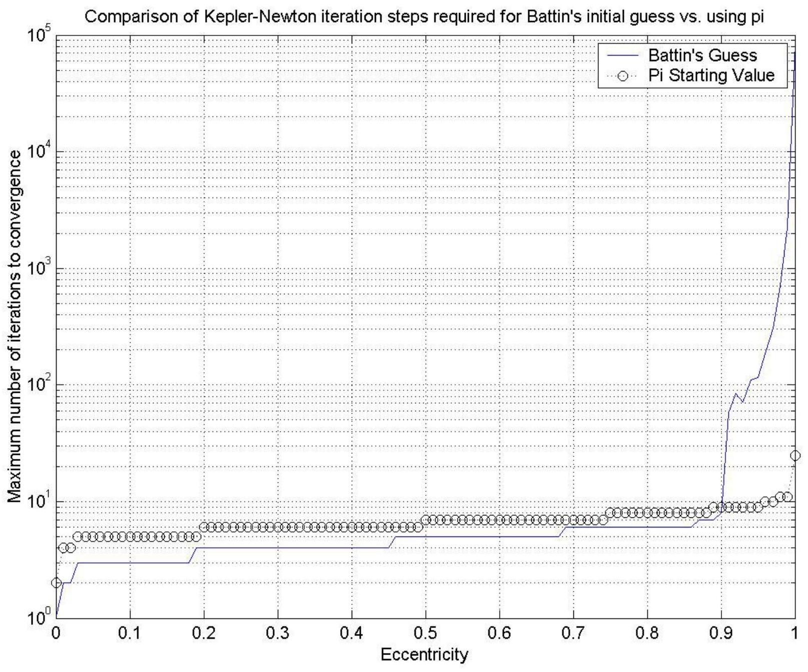


Figure 11. Comparison of the maximum number of iterations required for convergence when using Battin's approximation as an initial guess versus using π.

It was also desired to know the approximate computational savings achieved by using Battin's initial guess, when appropriate, as compared to simply using a value of π for the initial guess in all scenarios. Therefore, a new logic was implemented, called modified Battin, that simply used a starting guess of π if e was greater than 0.9, but would otherwise use the Battin initial guess. The total number of iterations for convergence was calculated for E every 0.005 radians between 0 and 2π (every 0.3 degrees, or ~1250 iterations/orbit) at each value of eccentricity between 0 and 1. The result in the total number of iterations, and the number of iterations saved by using the modified Battin technique is shown in Figure 12. The figure clearly shows that implementing a scheme to utilize the Battin initial guess as much as possible can lead to significant saving in the total number of iterations required per orbit.

The results above assume that M and E have values between 0 and 2π. If the object of interest has traversed more than one complete orbits, factors of 2π should be removed from M (using a modulus function) before these calculations are performed. Failure to do so will lead to a large increase in the number of iterations needed for convergence.

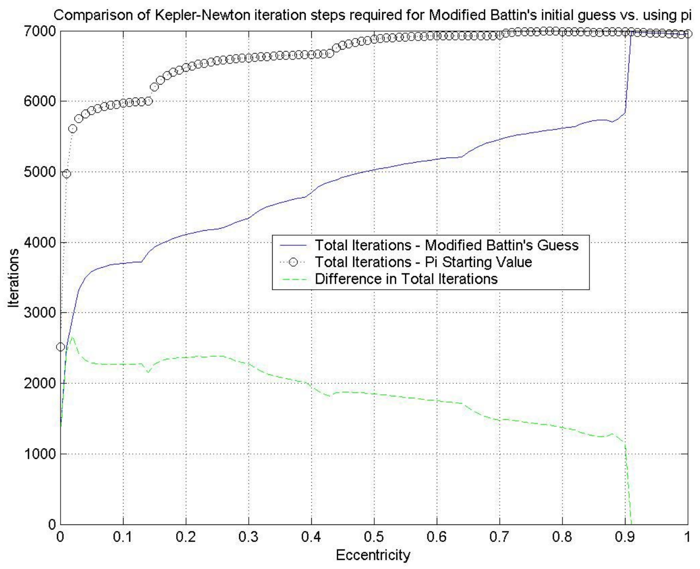


Figure 12. Total iteration steps required for convergence per orbit for Modified Battin versus using π.

It should be noted that both the KeplerBody and KeplerNewton routines measure the right ascension of the ascending node (Ω) from the inertial reference direction (as opposed to the prime meridian of the primary body.) The SEE source code and documentation may also refer to this quantity as the *longitude of the ascending node.* Applications including STK may make a distinction between the two, in which the longitude of the ascending node is referred to the prime meridian of the primary body rather than the inertial reference direction.

4.10 The Orrery

The SEE internal solar system simulator is referred to as the *Orrery.* This model may be used in the case where the user does not have or does not wish to use position versus time data generated by some external source. A list of planets and moons desired at application start-up is maintained in a configuration file.

In most cases the Orrery uses the dynamics model KeplerBody to determine the position and velocity of a planet or moon at a given time. An exception is the moon of Earth, which uses a special routine for improved accuracy. Section 4.10.1 describes this moon model.

The Orrery uses an application wide epoch and compares it to the current simulation time to generate a time delta for the current pass of the event loop. This time delta is used for calculating the position of the bodies along their orbits (e.g. true anomaly). The application epoch is also maintained in a configuration data file, read at application startup. Figure 13 shows a UML diagram of the classes comprising the Orrery.

4.10.1 Dynamics Model Class: Moon

`Moon` is a routine for determining the position and velocity of the Earth Moon (hereafter known as Moon or the Moon). The position information is based upon the low-precision formula for geocentric coordinates of the Moon, taken from the Astronomical Almanac, (pp. D46 for the Year 2000, and pp. D22 for the Year 2003)

It is stated that the errors from this technique will "rarely exceed 0.3° in ecliptic longitude (λ), 0.2° in ecliptic latitude (ϕ), and 0.2 Earth radii in distance (r)". The root-sum-square absolute positional error from this technique should then be between 2800 km (using an average moon orbit radius of 384,400 km) and 3400 km (using a maximum moon orbit radius of 403,620 km). These values are based on an Earth radius of 6378 km, and make small angle approximations for the positional error in ecliptic latitude and longitude.

The current moon dynamics model is still being reviewed. Preliminary documentation of the accuracy of this model as implemented in the SEE is provided in section 5.

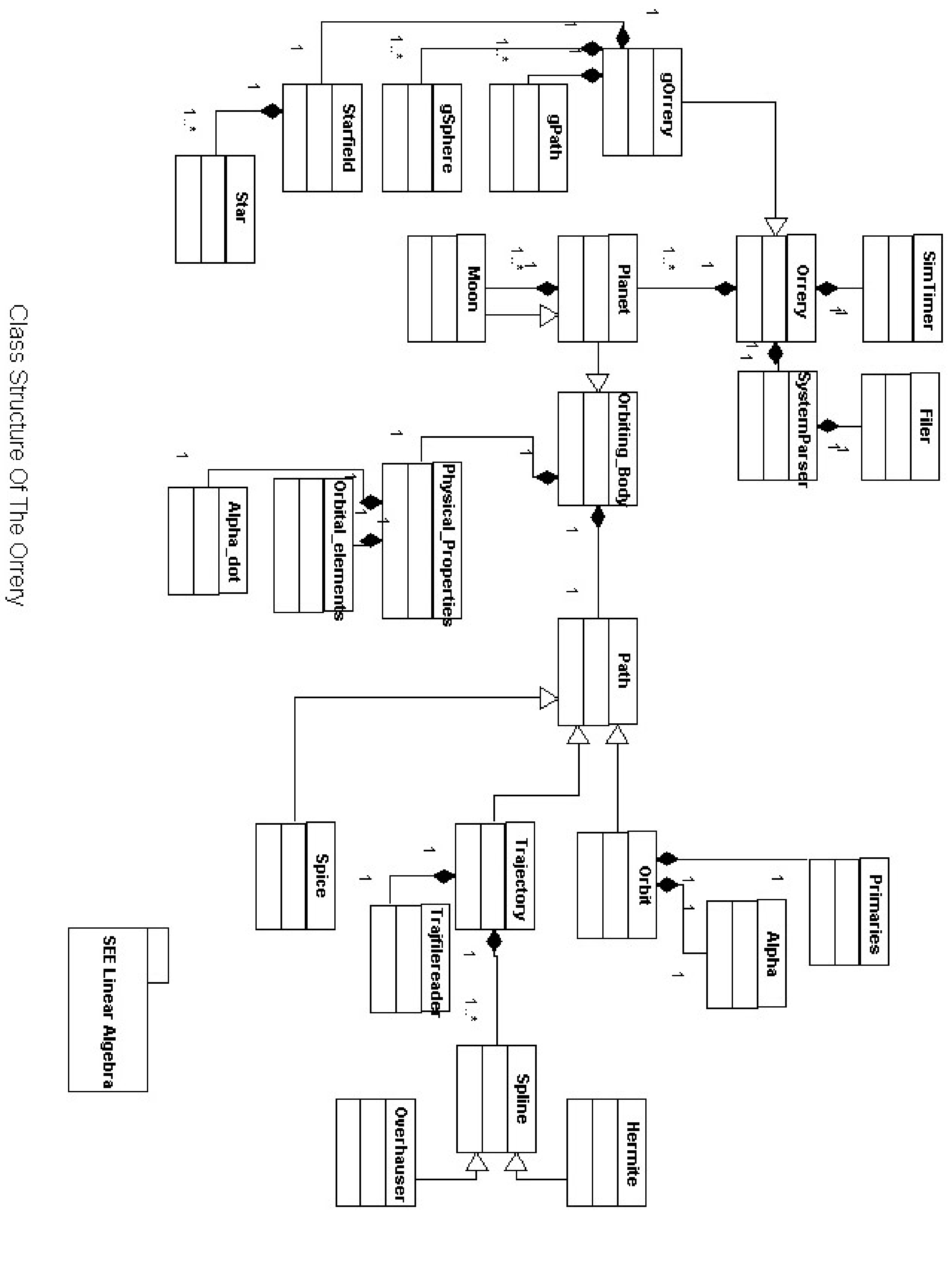


Figure 13: Orrery Class Structure

4.11 Crafts

A *craft* object in the SEE will normally be a man-made orbiting body such as a space station, satellite, or telescope. A complex craft such as the ISS may contain a number of features including articulating rigid bodies connected by joints (e.g. rotating solar panels), instruments, antennas, cameras, robots and visiting vehicles. The initial beta release of the SEE aims to model the spacecraft geometry and mass properties.

At a minimum an SEE craft requires a path in space over which it travels, and a geometric model for rendering. User-interface features will allow the user to generate orbits or trajectories by supplying orbital elements, position versus time data, and eventually telemetry streams. Geometric models will be loaded from files or assigned default primitives (sphere, cube) if the user desires. The SEE will include a library of ISS configurations based on the I-DEAS model files provided by the Systems Engineering, Modeling and Data Analysis (SEMDA) Laboratory at NASA Johnson Space Center.

The SEMDA Lab periodically publishes updates to the space station assembly sequence in a Design Analysis Cycle (DAC) report. This report contains estimates for the station mass and area properties for many of the planned station configurations, both with and without visiting vehicles. New I-DEAS model files are released with each DAC update.

The space station databases from SEMDA treat the space station assembly as a collection of connected rigid bodies. Figure 14, reproduced from the DAC 8 report, identifies the rigid bodies in the completed station. Each rigid body except the station core is connected to a parent body by a single pin joint (one degree of freedom). The station core has no parent, and is the only rigid body that does not contain a joint. Rigid bodies connected to the core are said to have an *alpha* joint. Rigid bodies connected to alpha joint bodies are said to have *beta* joints, and so on. The rigid body names and identification numbers remain the same for all configurations regardless of the assembly stage.

Each rigid body assembly is divided further into elements. An element is the smallest piece of the station for which there exists a single geometry file. The number of elements in each rigid body changes as new elements are added or moved as the assembly progresses.

This arrangement in which a craft consists of rigid bodies that are further divided into elements is reflected in the architecture of the C++ classes in the SEE (Figure 15).

Technical Monitor	Title		
T. Dawn/EA4/281-483-8095	International Space Station Program Design Analysis Cycle 8 Labeled Body Illustrations		
Approved By			
J. Posey/LM/281-333-7222			
Produced By	Contract		Item Number
Randall Hoerth	NAS9-19100 Science Engineering Analysis & Test		99-DR0130
	NASA Center / Division	Revision	Date
	JSC/Systems Engineering Office	Original	7/21/99

Notes:
1. Body 18 is not manifested at ISS assembly complete, but is depicted here.
2. Body 27, the Orbiter vehicle, is not shown.

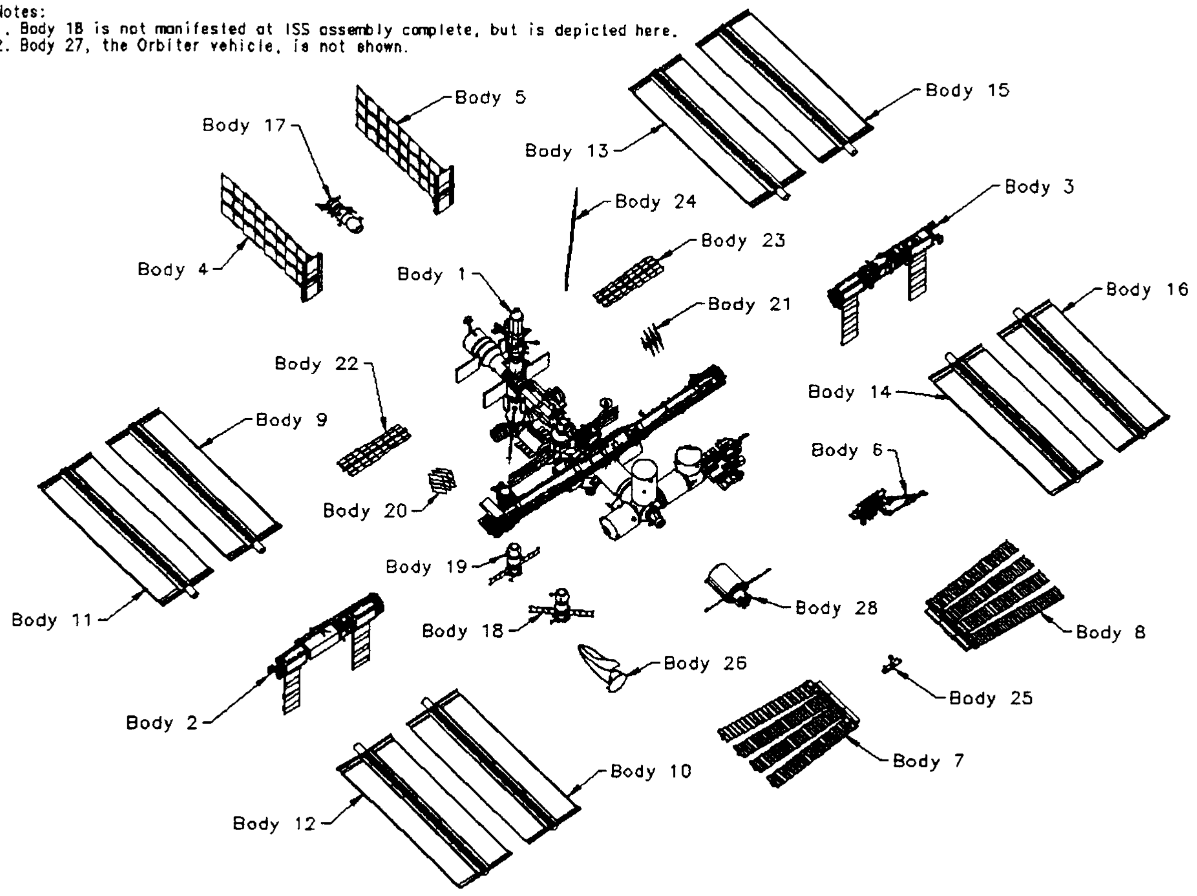


Figure 14: ISS Rigid Body Diagram (DAC8)

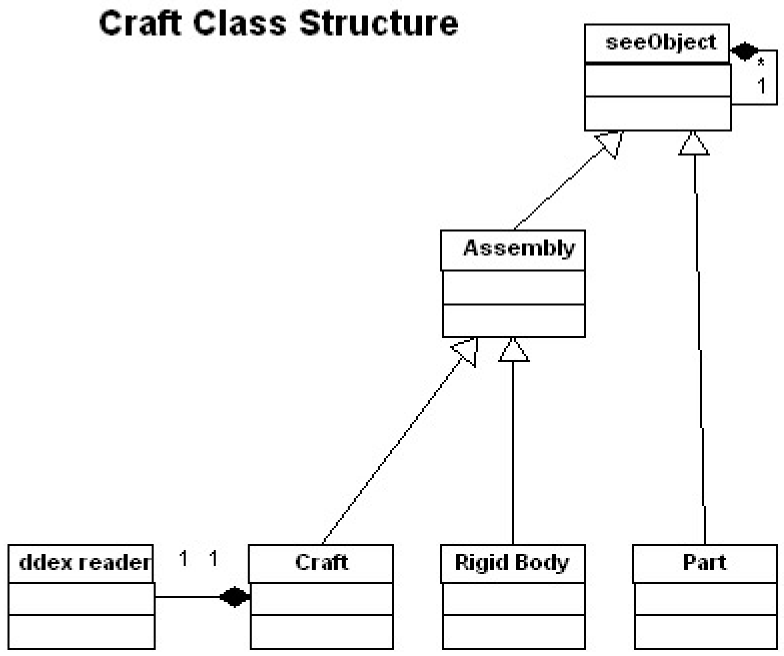


Figure 15: SEE Craft Classes

The `ddexreader,` appearing at the bottom left of Figure 15, is a class for parsing a file format (*.ddex) containing the assembly, mass properties and model file data for a given craft. An SEE utility program called the Dynamics Data Extractor (DDEX) is used to create the ddex file from an I-DEAS model database. I-DEAS is currently the only CAD package supported by the DDEX utility.

The ddex file can be thought of as the assembly specification for the spacecraft configuration. It contains the hierarchy of the rigid bodies, the position and orientation of each rigid body joint (known as the rigid body *pin*)[5], and the position and orientation of each element (known as the element *object matrix*). It also contains mass properties data, user notes, and an assembly tree reflecting the organization of the parts as they were stored in the original I-DEAS database. Element names contained in the ddex file are used by the application to locate associated vertex data files and to populate element selection lists for the GUI.

The DDEX utility also automatically creates geometry data (VRML files) from the I-DEAS database. These VRML (*.wrl) files are then converted to Performer binary files that are compatible with the Gizmo 3D engine.

Figure 16 shows how the SEE crafts objects relate to the reference frames in the scene graph. Note that the craft object hierarchy shows all rigid bodies as siblings on the tree, whereas the reference frame hierarchy is arranged to reflect how the rigid bodies are physically connected (e.g. the solar panel arrays are children of the Port Inboard Alpha truss). The organization of the reference frame hierarchy reflects the way the reference frame objects are organized in the scene graph, where leaf nodes will inherent the transforms of their parents. The organization of the craft objects reflects the way in which rigid bodies are created within I-DEAS. This organization will also be used in the selection interface design, where the rigid body names can aid the user in locating elements or assemblies of interest.

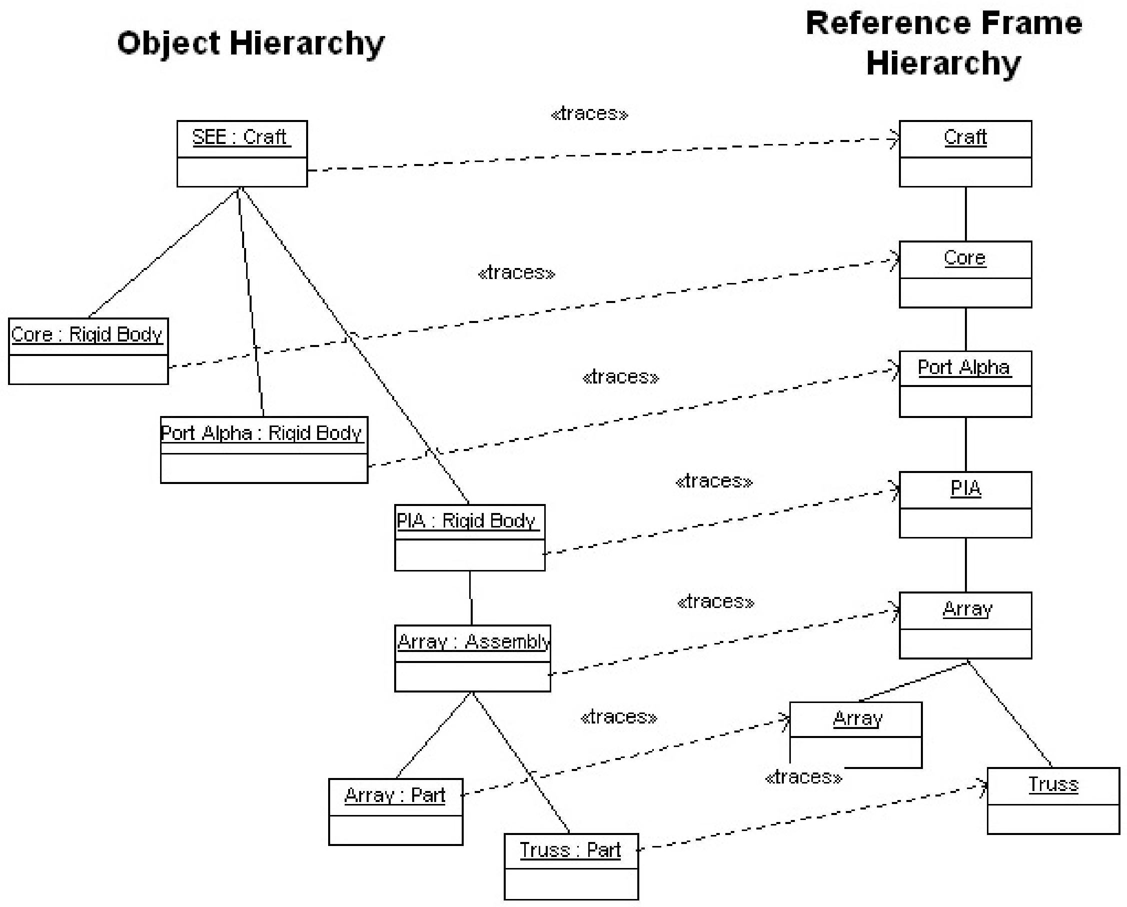


Figure 16: Craft Objects and Reference Frames

4.12 Visiting Vehicles

A *visiting vehicle* object in the SEE is used simulate and render a docking or undocking procedure. As a type of *craft*, visiting vehicles maintain all craft functionality. In addition, they contain the information necessary to locate the ports of the docking and target vehicles. In order to weaken the dependency of a visiting vehicle and its target vehicle, the final architecture for visiting vehicles will separate this information. All crafts will maintain the locations of their docking ports relative to the design reference frame. A *visiting vehicle manager* will coordinate the analysis of data (using software such as DOCKSIM), as well as the rendering of docking trajectories. The manager will require all visiting and target vehicles to register their port locations.

4.13 Collision Detection

Collision and proximity analysis is available in the SEE between any two crafts. The collision analysis determines at any given time step or series of time steps if any two objects have intersecting geometry. The proximity analysis indicates the minimum distance between any two objects.

These features are implemented in the SEE via the PQP application programming interface. The PQP API was created by the Gamma Research Group at the University of North Carolina (gamma.cs.unc.edu). PQP provides the user the ability to check proximity, tolerances, and collision between polygonal based objects. There are no topological restrictions on the objects being tested, such as convex hulls and holes in the surface. Each object is treated as a polygon cloud. To use PQP, the software developer must create PQP objects for those models to be tested. This is done by providing the PQP software the vertex data for each of the polygons belonging to the collision object. This data is provided in the objects design reference frame. For the SEE, a subroutine was created to extract out all of the polygons for each of the graphical models currently loaded in the scene. Therefore, no knowledge of what type of model was loaded into the application is required. After all of the polygons have been added to the PQP Model, the developer can call the PQP libraries and test for proximity distances, tolerance violations, and collisions. For each of these tests, the test requires the user to provide a position and orientation for each part to be tested.

A proximity test is an exhaustive test of all polygons of each model being analyzed. Upon completing this test, the PQP libraries provide the developer with the point in each model that represents the closest distance between the objects. This data is provided in local coordinates of each model. For a tolerance test, the user can ask the PQP software to check to see if the distance between two objects violates a given user specified tolerance distance. This test is similar to the proximity test, except that if a distance is found less than the tolerance, the PQP libraries stop checking for other closer distances. The two points that form the distance violating the tolerance distance can be queried from the libraries. For a collision detection test, the PQP libraries check to see if any polygons on the two models are intersecting. The user can specify one of two options for this test. The user can tell the PQP libraries to stop checking for collisions once one collision is found, or the PQP libraries can be told to continue to check and extract all collisions. A query for if a collision occurred and a list of colliding polygons can be obtained from PQP. Again, the data is provided back in local coordinates to the collision models.

Preliminary testing of the PQP software in the SEE indicates that even for scenes with polygon counts in the 100,000 to 200,000 range the collision detection algorithm will run at reasonable frame rates.

4.14 Cameras, Rendering Windows, and Navigation

When a user asks the SEE to create a new graphical (or rendering) window, the program must create a virtual camera, located and oriented within the scene, and connect the camera view to the screen space occupied by the rendering window. The rendering window also accepts mouse and keyboard inputs to allow the user to navigate. Thus the classes that control the graphical (or rendering) window, the virtual camera, and navigation through the scene are closely related. Figure 17 depicts the relationship of the main classes for these functions.

The rendering window is created using class `GWindow`, a simple Qt window. Requesting a new `GWindow` object creates a new camera. The `GWindow` houses a `GWidget`, the Qt rendering widget. The `GWidget` class forms the primary handshake between Qt and Gizmo3D. It is a Qt widget that contains the Gizmo3D rendering object, a `gzWindow`. In the construction of a `GWidget`, a `Camera` object is assigned, paralleling the Gizmo3D `gzCamera` assigned to the `gzWindow`. The `Camera` object maintains a link to the `gzCamera`, and is responsible for updating the `gzCamera`'s settings on a per frame basis.

One or more `Scene` objects will be instantiated within the application. A `Scene` object is responsible for monitoring the construction of a Gizmo3D scene graph with a `gzScene` root node. A `Scene` must be assigned to a `Camera` before rendering with a call to `setScene`. A Camera's scene may be changed with repeated calls to `setScene`.

Camera navigation is controlled by means of an `MVTransformer` object. Subclasses of `MVTransformer` control specific navigational behavior (such as trackball or fly modes) and accept keyboard and mouse events. Currently two transformer classes are available, `MVKMFlyTransformer` and `MVKMTrackballTransformer`. These classes are instantiated within a `GWidget` object, which passes Qt keyboard and mouse events to the transformer. The `GWidget` is also responsible for determining the current transformer to be used by the `Camera`. The transformer translates keyboard and mouse events into a position and orientation matrix for the camera. The `Camera` uses this matrix to update the Gizmo3D `gzCamera`.

`Cameras` are deleted when the parent `GWindow` is deleted. The `GWindow` is set to have a destructive close, so that close events result in the deletion of the `GWindow`, `GWidget`, `Camera`, transformers, and corresponding Gizmo3D `gzWindow` and `gzCamera`. The `Scene` and `gzScene` objects persist.

A snapshot of the active camera view can be saved in several file formats including .bmp, .jpg and .png. Image capturing within the SEE is a two-phase process. The first phase captures the rendered scene into memory by use of the Gizmo `gzImageRender` class. In effect, a rendering pass of the scene graph is made with the resulting image stored in memory (as opposed to being displayed on a monitor). In order to handle depth buffering issues, the `gzImageRender` class has been subclassed to perform multiple rendering passes, far to near. In the second phase of the process, the resultant image is transferred to a `QPixmap` object, an SEE "stamp" is applied to the image, and the image is saved to disk in the desired graphics format.

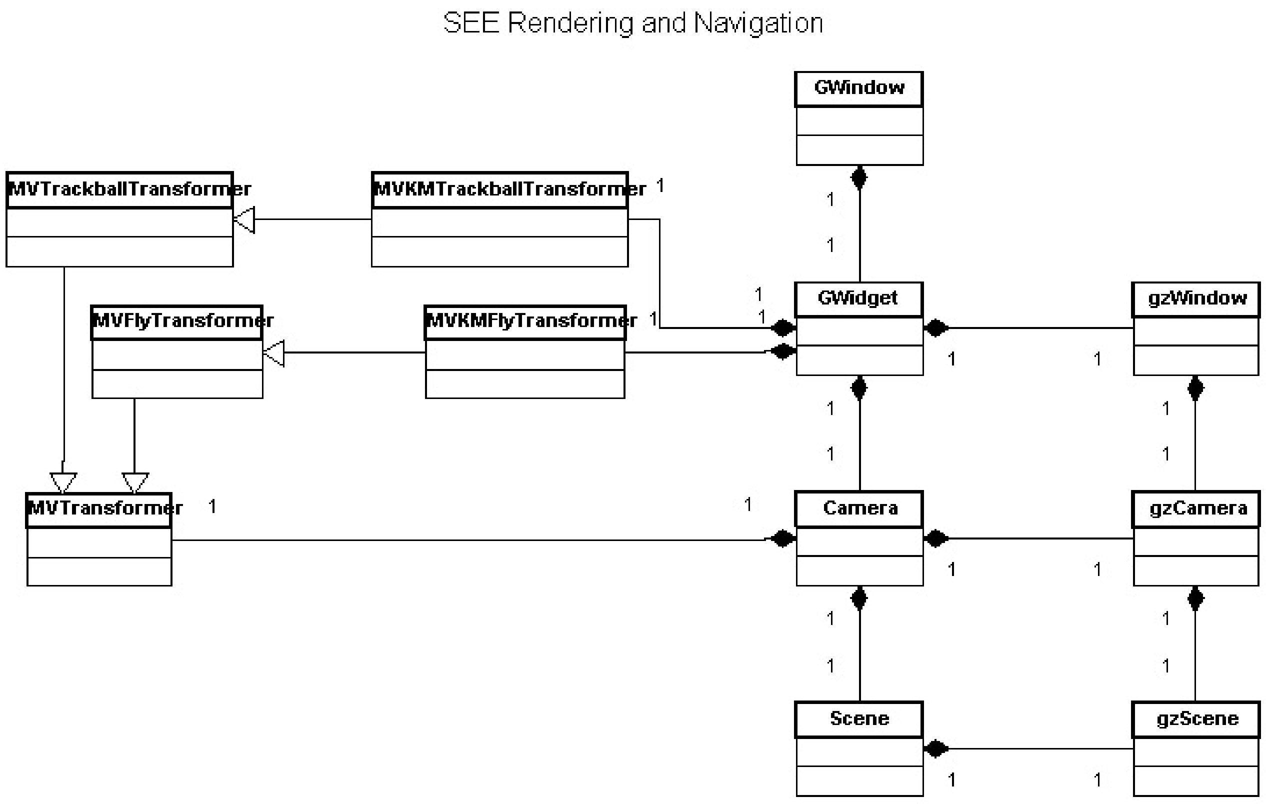


Figure 17: Rendering and Navigation Classes

4.15 The Simulation Time System

The simulation time is stored internally as a Julian Date in the `SEEdate` class. Two double precision floating point variables are used to store this value. The first double represents the number of days elapsed since 4713 BC January 1, Greenwich noon, truncated to the nearest whole number of minutes. The second double represents the number of seconds, between zero to sixty, beyond the current minute. Stored in this way, the date and time can be resolved to roughly 1.0e-13 seconds for years between 4713 BC and 22,666 A.D. Any time dependent variables in the SEE will be updated once per event loop, based on the Julian Date for the current frame. For algorithms that operate on an elapsed time (such as the Kepler orbit creation routines), the delta-t is computed by comparing the current Julian Date is to the epoch being used for that algorithm.

All *delta* times are returned in seconds only. This limits the resolution on the addition or subtraction of time from a selected date depending on the value of the delta. For example, when adding 946728000.000000 seconds (~30 years) to a selected date, the available precision is limited to 1.0e-6 seconds by the 15 available digits in the seconds field.

The `SEEdate` class overloads the appropriate C++ math operators to allow differencing dates, adding or subtracting times from the selected date, and comparing dates. Input and output stream operators are also overloaded to allow quick access to string versions of dates at any a configurable precision. Time zones are supported by the `SEEdate` class but are not used in the current version of the SEE (all time zones are entered and reported as UTC).

When the simulation time is running, the rate of the passage of simulation time is configurable by the user. The precision of the SEE time system allows the time rate to span between one one-thousandth real time and thousands of years per second. The simulation time for a given frame is calculated by polling the operating system clock at successive passes through the event loop, multiplying this elapsed real time by the user specified time rate multiplier, and adding it to the simulation time for the previous frame (Figure 18).

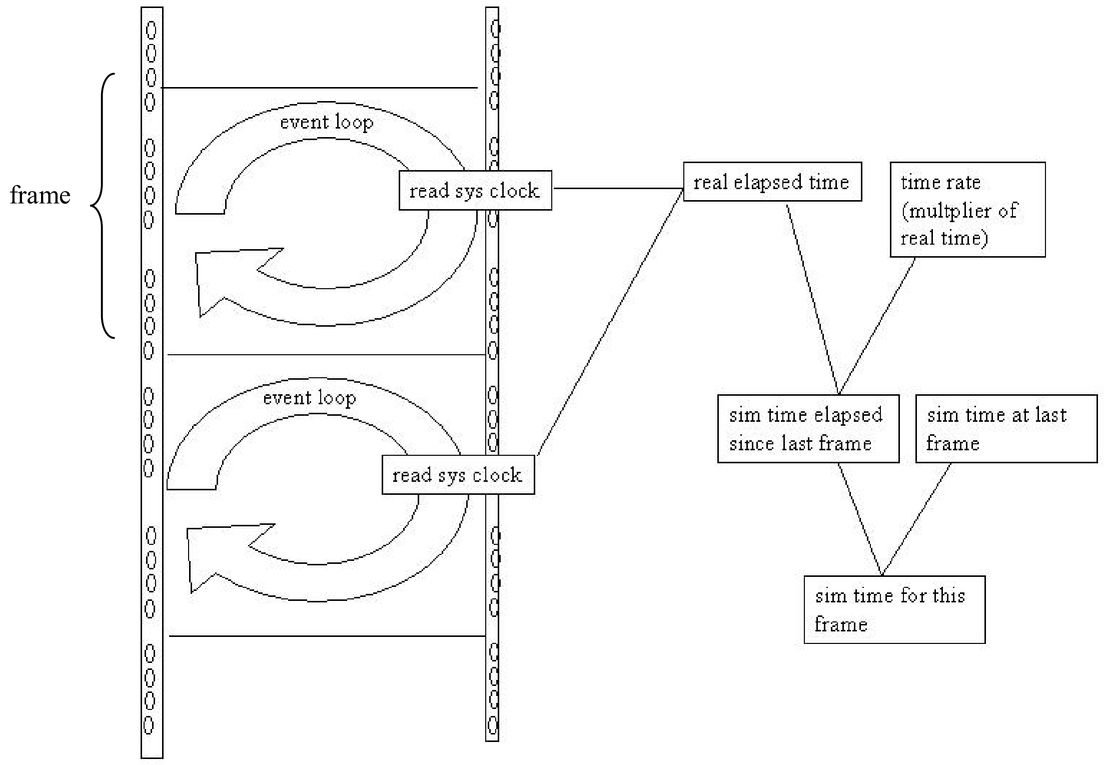


Figure 18: Preparing the Simulation Time

4.16 Building the Mission

The collection of objects in a given scene is referred to in the SEE as a "mission". A mission will contain a model of the solar system containing a model of the Sun and any additional planets, moons, crafts and other SEE objects added by the user. Data files containing libraries of objects that can be added, deleted, or modified during the application session are provided as part of the SEE and are referred to as "stock" objects. After a stock object is added to the scene the user may modify the parameters that describe that object and export the modified version as a "custom" object. These custom objects are saved in a user-writeable directory and are available for import to other missions.

All of the information required for the stock planets is hard-coded into a `std::map` member of class `SolarSystem`. This `std::map` stores the name, parent, number of children, source type, texture map, texture offset, and number of points for the orbit path. The hard-coding of this information enables the setup of the solar system using only the names of the desired planets and moons. This listing is placed in the file *solar_system.ini* included with each mission. An example of the file is shown below:

Example of a solar_system.ini file:

```
[SOLAR SYSTEM]

[STOCK PLANETS]
EARTH
MOON
JUPITER
IO
EUROPA
#End of List#

[CUSTOM PLANETS]
Custom Planet = Mars2
```

The custom planet "Mars2" will be described further in the object subdirectory of the same name. In this subdirectory the file with the planet name and a ".psys" extension will list properties of the planet and its moons. Note that moons cannot be imported or exported separately from planets – they are always stored together as a "planetary system". An example file is shown below:

File Mars2.psys:

```
[DESCRIPTION]
Name = Mars2
Notes = Custom Mars System
Primary = Sun
Number of Children = 2

[DESIGN REF FRAME]
Name = J2000
Parent = Ecliptic
Base_Source = Kepler
Base_Source_File = kepler4.ref

[PYHSICAL PROPERTIES]
Mass = 0.64185e24
Equatorial radius = 3394.0

[GRAPHICS]
Model = sphere
Tex Map = mars/marsx.png
Tex Offset = -90.0

[PATH]
Red = 0.3294118
Green = 0.000000
Blue = 1.000000
orbit_scale=1.0
num_pts=100

=========================
[CHILDREN]
=========================
Name = Phobos
Notes = Custom Mars System
Primary = Mars2
[DESIGN REF FRAME]
Name = J2000
Parent = Equatorial
Base_Source = Kepler
Base_Source_File = kepler2.ref
[PYHSICAL PROPERTIES]
Mass = 0.0000000106e24
Equatorial radius = 13.0
[GRAPHICS]
Model = sphere
Tex Map = mars/phobos/phobosx.png
Tex Offset = -90.0
[PATH]
Red = 0.5
Green = 0.000000
Blue = 1.000000
orbit_scale=1.0
num_pts=100
[END OF ENTRY]
```

The `systemParser` class handles the reading of this file and stores the desired planets and moons in a `std::vector`. This `std::vector` is then iterated and the required information is extracted out of the `std::map` member of the `SolarSystem` using the object's name. This creates the solar `struct` in the `systemParser` which is used to retrieve the ephemeris data out of the *solar_system_data.ini* file.

A wizard interface is provided at application launch to handle the retrieval of a saved mission or to create a new one to be used in the current session. A mission is stored as a directory tree in a user-writeable directory path specified with the environment variable `SEE_USER`. An example of a mission directory tree is shown below:

Example Mission Directory Tree:

```
c069_12a-dac8
      |
      c069_12a
                  |
                  craft.dat
                  c069_12a.ddex
                  default.man
                  kepler.ref
      |
      solarsystem
                  |
                  solar_system.ini
      |
      mission.dat
```

All missions are loaded from a temporary subdirectory "/*working*" under `SEE_USER`. Retrieving a saved mission is implemented by replacing the current *working* directory with the contents of the directory tree for the incoming mission. The file *mission.dat* located at the top level of the mission directory tree contains links to all other optional files needed to load the requested objects. An example file is shown below:

Sample Mission.dat file:

```
[MISSION]
Title = ISS 12a (DAC 8)
Description = Default mission for 12a

Start Year = 2003
Start Month = 4
Start Day = 10
Start Time = 12:00:00

[CRAFTS]
Craft = c069_12a
```

New crafts may be added to the current mission interactively using the *New Craft* wizard interface. When this new craft is loaded the system will go through a check to enforce name uniqueness among the crafts already in the scene. Initially the craft directory is set to match the craft name entered into the wizard by the user. To make sure that the adding of this craft does not overwrite any of the existing crafts already in the *working* directory, it checks to see if this craft directory is unique. If its not unique, it is appended with a number before it is copied into the working directory. The next check for uniqueness occurs in class `craft` where the craft's name is checked for uniqueness. If its name is not unique, the name is appended with a number. Once the craft name and directory have gone through their uniqueness checks, the craft is then added to the `mission.dat` file of the working mission. This enables the mission with the new craft to be resumed after an exit or saved during a session.

4.17 Bookmarks

The *Bookmarks* feature of the SEE enables the user to save and recall some of the environment settings in order to allow the given view to be easily re-constructed. As of version beta 0.40, these environment settings include the simulation time, the scaling factor that has been applied to any object in the scene, and the icon objects that have been added to the scene.

The elemental class of the bookmarks feature is a virtual class called `marker`. The objects of the marker subclasses (`scalemarker`, `timemarker`, `iconmarker`) contain the data to be saved and retrieved (e.g. an object scale, simulation time or icon state).   An object scale marker, for example, may contain scaling parameters for some or all of the objects in the current mission. Collections of `Markers` are assembled into objects of class `Bookmark`. A `Bookmark` object may contain any number of marker objects which need not be of the same type.

Routines for creating, deleting, loading, saving and otherwise managing the user's bookmarks is encapsulated in the class `BookmarkManager`. Separate *activate* functions are provided in the `BookmarkManager` to retrieve and use the information in the three types of markers. Also in `BookmarkManager` are special functions for finding the nearest time marker, the nearest marker ahead of the current time, and the nearest prior time marker.

4.18 Analysis Tools

The SEE provides several analysis features for extracting quantitative information from the current scene. While the specific implementation of these features is dependant on the type of analysis, the general architecture for a given analysis tool should follow one of two patterns. For analyses that require data to be gathered over the course of a selected time span, the SEE analysis routine enables a flag that tells the event loop to make a pass through the data-gathering function of the corresponding analysis code with each frame update. For analyses that are time-independent (e.g. the calculation of area properties for a selected craft) the SEE event loop is halted until the analysis is finished.

No commands are accepted from the user in this mode except those presented by any status dialog boxes displayed by the analysis routine (e.g. a ‘Cancel’ button to abort the analysis). This version of the SEE does not utilize multiple program threads, so events in the background may not continue while the SEE event loop is halted.

The Collision/Proximity analysis, Line-of-Sight analysis, Dynamics Report, CAPS Full Survey, and ARCD are all examples of time-dependent analyses. The time-dependant analyses may be further classified into two types, distinguished by whether the routines require a fixed-time-step analysis or a current-frame analysis. A fixed-time-step analysis requires the SEE to collect data over a specified range of simulation time at specified intervals (time steps). When this type of analysis begins, the simulation time jumps to the required value and a flag in the event loop is enabled to indicate that additional information is being requested by an analysis function. When the current frame is complete, the simulation time is automatically advanced by one analysis time step. The process completes when the analysis stop time is reached. Data files and summary reports containing the results of the simulation can be viewed once the analysis has finished. During a fixed-time-step analysis the user is not able to issue time navigation commands. Depending on the specific analysis code, the SEE simulation time after the analysis is complete may be left at the last analysis time step, the first analysis time step, or the time at which the analysis was initially launched. The Dynamics Report, CAPS Full Survey and ARCD are time-stepped analyses. The Collision/Proximity tool has both a time-stepped mode and a current-frame mode.

A current-frame time-dependent analysis such as the Line-of-Sight tool or the ‘current-frame mode’ of the Collision/Proximity tool does not take over control of the SEE simulation time. The user has full time and space navigation capability. However, since the data request by the analysis routine must be re-calculated every frame, the application frame rate may be impacted when an analysis of this type is enabled.

The *Area Report* function is an example of a time independent analysis routine. Once a request to run the area report has been made, the event loop is suspended until the operation is complete. To display the current completion status of the command, a progress bar is displayed along with a cancel button to allow the user to abort. Because the event loop is not active, the QT *processEvents* command must be called at regular intervals in the code that is used to collect the area data. This command allows the progress bar to be re-drawn and the cancel command to be processed if one was issued. All procedures that suspend the event loop for significant periods of time should provide a progress status to indicate to the user that the SEE application is still functioning. If possible a cancel or abort opportunity should also be provided.

4.19 Jet Plume Visualization

Thruster jet plumes may be visualized in the SEE for crafts that contain jet firing data. The jets associated with a craft or visiting vehicle are described by a text file located in the craft subdirectory. The file describes the color, size, shape, location and name of each jet on the craft. Each jet description may optionally contain the name and format of a jet firing history file. Currently supported jet firing history formats include the Docksim–RCS format, and a "Compact" format. The Docksim-RCS format contains firing schedules for a number of thrusters in a single file. The thruster ID number is used as an identifier in this case to connect the jets to the correct firing data. The Compact format contains a single jet firing history data set consisting of two columns, the jet firing start times and the firing durations. The jets on a given craft can each be independently assigned a jet firing history in either of the currently available formats.
The plume visualization controls are available in a large dialog box format that provides automatic help text, or in a compact dockable toolbar mode. The controls allow the user to select a minimum opacity level that will be used for all jets to draw the plume cone when the thruster is not firing. This mode can be used to examine the location of all the thrusters. When the thruster is firing, the visualization can be done in one of two modes. In each mode the opacity level will be increased from the user selected minimum according to an algorithm designed to allow meaningful visualizations at various simulation rates. These algorithms are designed to correct the problems caused when the elapsed simulation time between subsequent rendered frames is much larger than the typical thruster firing duration. This is frequently the case since jet minimum on-times are usually in the millisecond range, whereas docking maneuvers and other craft motions typically occur over minutes or hours. Using a simple "on-or-off" algorithm to draw the thruster cone at the rendered frame may produce poor results under these conditions, e.g. making a constant duty cycle appear erratic, making high duty cycles appear as steady-state firing, and missing low duty cycles altogether. The solution implemented here utilizes a time sampling window centered on the current frame to increase the jet opacity according to the amount of jet on-time that occurred in the window. When the jet fires through the entire sampling window, the jet opacity is set to 100%.
In "automatic sampling" mode, the size of the window is automatically adjusted according to the current simulation time rate. Specifically, increasing the time rate will grow the sampling window so that short pulses will not be missed. Slowing the time rate will shrink the window, allowing the jet firing times to be resolved more accurately. In "manual" sampling mode, the sampling window size remains constant until the user adjusts it. This mode is particularly effective when the simulation time is stopped. Adjusting the sampling window when time is stopped will reveal which thrusters were most active in the vicinity of the current frame.

Class `FOVManager` is used to read the jet information associated with a craft, which is always stored in the "fovs.dat" file in the craft directory. `FOVManager` also instantiates the `Thruster` objects and connects the thruster to the craft in the object hierarchy and reference frame hierarchy. Class `Thruster` is derived from class `FieldOfViewCone` and adds functions for handling the jet firing history visualization techniques.

4.20 Plotting

The SEE plotting routines make use of the Qwt widget libraries to graphically display the plot data, and are otherwise implemented like a time-stepped analysis tool. The user sets up the desired plot parameters via the wizard interface, and selects *Finish* to start the plot data collection procedure. For all plots except Jet Firing Histories, clicking the *Finish* button sets a flag that directs the SEE to take control of the simulation time and collect the desired plot data. The data may include one or more vectors of time-dependent parameters. Each of these vectors and a vector containing the Julian date of each sampled time is sent by the `PlotManager` object to the `PlotDialog` widget. In class `PlotDialog` the vectors are converted to plot lines utilizing the Qwt graphing widgets. For the Jet Firing History case, data is not sampled inside the event loop but is read directly from the jet history data stored in the `Thruster` objects.

In the plot display the current simulation time is always marked by a red vertical line unless the plot window does not span the current time.

Note that the plot data cannot be changed after the plot has been drawn. If the user makes changes to the mission after a plot has been made (e.g. changed the attitude parameters of a spacecraft after the yaw-pitch-roll sequence has been plotted), these changes will not be reflected on an existing plot. New plots will of course make use of all current data that exists before the plotting wizard is completed.

4.21 Macros

Macros are a convenient means for performing repetitive tasks within the SEE application. Scenarios in which the user needs to repeatedly create lines-of-sight or run thousands of collision analyses in batch can benefit from the use of macros. In brief, macros are created and managed and activated through the `MacroManager` class. Even though it is the primary container, the `Macro` class is extremely simple. It contains a list of `Commands`, a method for adding `Commands`, and an output operator. The `Command` class is the workhorse of macro implementation. This base class is subclassed in one-to-one correspondence with the SEE managers. Each `Command` object stores an action to be implemented by some SEE manager and all of the parameters required by that action. When a macro is activated, the `MacroManager` is responsible for sending each of these commands to the appropriate manager within the SEE application. In turn, each manager is responsible for activating the action issued by the command.

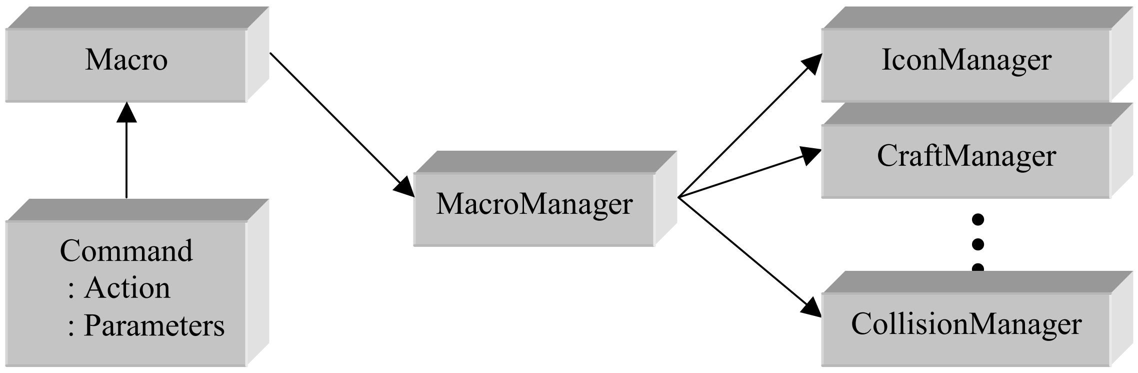


Figure 19: Macro Classes

The Command Class

Each SEE manager should have a corresponding command class that inherits the `Command` class. The subclass must, in its constructor, register all actions handled by the manager along with a list of parameters for each action. A command is created by setting the desired action and appropriate parameters. The subclass typically need only implement the methods for saving or retrieving parameters whose values are to be chosen from a list. Most input, output and parameter retrieval is handled by the `Command` base class. Note that the parameter values are stored using void pointers and type information. Thus any subclass accessing the parameters must perform dynamic type casting.

Creating a Macro

Loading, saving, creating, and activating macros are all within the purview of the `MacroManager` class. To create a macro, a name must be supplied to the `MacroManager's createMacro` command. A pointer to an empty macro is returned to the caller. Commands are created by sending a manager type (or category) to the `MacroManager's createCommand` method. A pointer to an empty command is returned to the caller. The command's action is set and the appropriate parameters are stored within the command. The command is then added to the macro. The order in which commands are added will determine the order in which they are executed when the macro is activated.

Activating a Macro

Sending a name to the `MacroManager's activateMacro` command will start the playing of any macro matching that name. The `MacroManager` steps through the macro's list of commands in order. The command type is identified and the command is sent to the appropriate SEE manager's `doCommand` method. The SEE manager parses the command's action and parameters and performs the required duties. When the action is complete, the SEE manager emits a `commandDone` signal. The `MacroManager` picks up this signal and the next command in the macro is issued. The use of signals allows the macro to execute in a linear fashion while maintaining the main event loop.

4.22 Movie Capture

The recording of AVI movie files is a two-phase process, preparing the frames (images) and writing the movie file. An AVI file consists of a sequence of still images which when played in rapid succession produce an animated effect. In the SEE application, movie images are captured using the same device as still image captures, a `gzImageRender` object. This object acts as a virtual camera, the only difference being that results are rendered into memory rather than to the screen. The `gzImageRender` object renders the current scene to a `gzImage_RGBA_8` object whose width and height match that of the current screen window. The pixel color information is then extracted from the `gzImage_RGBA_8` object and stored in a BGRA byte buffer for movie processing.

The creation of AVI movie files uses platform dependent code. The ability to record AVI files is currently restricted to the Win32 platform. The platform dependent code is wrapped within the Movie object so that the SEE application my interface with Movie object in a platform independent fashion. The MSVC++ compiler on Win32 platforms supplies the library that generates the AVI files. An AVI library will be needed for the Unix platforms.

The three steps for creating an AVI file are opening the file for writing, writing each frame to the file, and then closing the file. The Movie object's open method requires an output file name, the size of the images to be added, the playback rate, and whether or not the file is to be compressed. If the file is to be compressed, the Cinepak compression codec is used and a compression rate should be supplied. After the movie file has been opened, each frame is added to the file by sending the `Movie` object a buffer of pixel color data corresponding to the size specified in the open command. After all the frames have been added, a simple close method should be called.

The `Movie` object also offers a static method for compressing uncompressed movie files. The name of an existing uncompressed AVI file and the output file name should be supplied as well as the level of compression desired. The routine will read in the uncompressed file and produce a file compressed using the Cinepak codec. If a Qt progress bar is supplied, it will be updated to show the progress of the compression. The SEE application uses the `Movie` object's compress method within a Qt thread so that the main application may continue to operate during the compression of a file. Progress is displayed on the right end of the main window's status bar.

4.23 Online Help

Online help features are integrated in the SEE by utilizing the Qt Assistant, a help browser freely distributed within the Qt development package. The Qt Assistant is customized by the creation of an Assitant document profile, a file format which is fully documented in the Qt online manuals. A help window is launched from within the SEE by instantiating a QAssitantClient object and specifying a URL pointing to the file to be displayed.

All help windows are launched from the MainWindow object. Any dialog that needs to activate a help window does so by emitting a signal which must be connected to the MainWindow's openHelpAssistant( ) slot. The QAssistantClient will launch the Qt Assistant and establish communication with that process. Future requests will be displayed in the same process window if the user has not closed it; otherwise, a new process will be launched.

A given SEE dialog is connected to a specific URL in the HelpDesk object. The HelpDesk maintains a map of dialog names and URLs. Currently, all help pages are extracted from the SEE User Guide. Should the user guide be significantly altered, then the HelpDesk must reflect those changes.

5 VERIFICATION

The results of recent tests of the SEE dynamics routines and planned tests are described here. These tests should be performed when modifications to the SEE dynamics codes are made.

5.1 Test Case 1

Objective:

Perform a visual check of a spacecraft object position over the Earth within the SEE spacecraft to known spacecraft position over Earth. This should verify that a craft placed in Earth orbit with known orbital elements is accurately placed within the SEE environment, relative to the latitude/longitude grid on the SEE Earth. The approach would be to put the orbit elements for a known object into both the SEE and Satellite Tool Kit (STK) software, and perform visual and measured comparisons for the object latitude and longitude at the time for which the orbit elements were specified.

The spacecraft chosen was the International Space Station (ISS). The position information for ISS was taken from published two line element sets (TLE) published by NORAD and used for object tracking over short durations (TLE found at 'http://www.hq.nasa.gov/osf/station/viewing/issvis.html').
The TLE set for ISS at 00:17am EST on 12/03/2002 were obtained. TLE information is specified relative to the true equator, mean ecliptic of epoch (TEME) coordinate frame. The data was:

```
ISS
1 25544U 98067A   02337.22028935  .00190497  00000-0  25489-2 0  4351
2 25544  51.6340 284.2634 0003290 235.0992 330.7125 15.56838241230339

Name......................................ISS
NORAD ID#.................................25544
Epoch Year................................2002
Epoch Day.................................337.2203 = 12/3/02  00:17am
EST
Mean Altitude (km)........................396.768
Period (min)..............................92.49
Apogee (km)...............................398.997
Perigee (km)..............................394.539
Inclination (degrees).....................51.634
Right Ascension of Ascending
  Node (RAAN, degrees)....................284.2634
Eccentricity..............................0.000329
Argument of Perigee (degrees).............235.0992
Mean Anomaly (degrees)....................330.7125
Mean Motion (revs. per 24-hr. day)........15.56838
Decay Rate................................0.00190497
Epoch Revolution (since Zarya launch).....23033
Element Set#..............................435
Visible up to Latitude (degrees)..........71.3
```

The above epoch is 05:17:13 am UT, or 2452611.7203 Julian date.

The TLE information for the TEME coordinate frame was converted to J2000 orbit elements by utilizing the menus within the STK toolkit. The resulting J2000 orbital elements for the ISS used were:

Time (UTCG)	3 Dec 2002 05:17:13.00
Semi-major Axis (km)	6774.904955
Eccentricity	0.000329
Inclination (deg)	51.620
RAAN (deg)	284.228
Arg of Perigee (deg)	235.096
Mean Anomaly (deg)	330.712

Information extracted from STK for the ISS with the above orbital elements was:

Time(UTCG)	3 Dec 2002 05:17:13.00
Latitude (deg North)	-20.064
Longitude (deg East)	-30.214
Altitude (km)	397.323361

```
Spacecraft just East off the coast of South America.
```

Results from the SEE using the J2000 orbit elements above were:

```
Spacecraft appears in approximately the same place as in the STK model.
Estimated Latitude (deg North) = -20 +/- 1.
Estimated Longitude (deg East) = -30 +/- 1
```

The latitude and longitude grid within the SEE has a resolution of 10 degrees between lines, therefore the estimated accuracy of measurements (done visually) was +/- 1 degree. The SEE currently doesn't contain a routine to report sub-object point latitude and longitude.

Additionally, the Moon position at time zero is given within the STK model to be:

Time (UTCG)	3 Dec 2002 05:17:13.00
Latitude (deg North)	-18.092
Longitude (deg East)	82.696
Altitude (km)	356475.066483

Results from the SEE for the Moon position at time zero were:

Time (UTCG)	3 Dec 2002 05:17:13.00
Latitude (deg North)	_____________
Longitude (deg East)	_____________
Altitude (km)	_____________

5.2 Test Case 2

Objective:

Estimate the accuracy of placement of the continental map and latitude/longitude grid on the Earth sphere relative to an external reference point, and relative to each other.

<u>Important notes about latitude and longitude estimations within SEE:</u>

The accuracy of the estimation in latitude and longitude is limited by the accuracy of two separate steps that have been used to position the continents and lat-long grid on the Earth at a reference time:

1. How accurately the continental texture map features line up with a reference line (such as a line connecting the Earth and Sun) at a specific date and specific time as compared to a reference measurement. Checks have been performed to test the current alignment, and the results are summarized below.
2. How accurately the lat-long grid is located relative to the texture map of the continents that is used , i.e. how well the 0 degree longitude line aligns with the apparent position of Greenwich, England, and how well the 0 degree latitude line aligns with an appropriate feature on the continents. This is dependent on the detail of the grid (number of pixels wide each lat-long line consists of) and the accuracy in estimation of the approximate location of the reference points (Greenwich, etc.) on the continent texture map.

The lat-long grid was aligned visually by D. Murphy and D. Cornelius on top of the Earth texture map being used by the SEE on 12/5/2002. A new Earth texture map with lat-long grid included was then created. This new texture map was used to make the estimations in lat-long that are quoted for the test cases.

Accuracy of the lat-long to continent texture map alignment:

1. Within the SEE texture map, each lat-long line is 3 pixels wide. The assumption was made that the middle pixel was the center of the line, leading to a line placement accuracy of ½ pixel. The position of reference locations within the continent texture map was estimated to be within 1 pixel. The texture map of the entire globe consists of a picture that is 1024 pixels wide and 512 pixels tall, a total placement error of +/- 1.5 pixels, or 0.53 degrees in both longitude and latitude.

Accuracy of the placement of the texture map on the Earth sphere relative to a reference object:

The sub-solar point on the Earth for various dates was used to estimate the accuracy of the rotational alignment of the Earth with a reference object. Multiple dates were selected between 1 Jan 1980 and 1 Jan 2040. Comparison of the time of prime meridian alignment with the sub-solar point was made between the SEE and STK. Results are summarized in Table 1 below.

Table 1. Estimated time of Sub-Solar Point Alignment with the Prime Meridian for SEE versus STK.

Date	SEE		STK		
	Time of Prime Meridian Alignment with Sub-solar Point (hh:mm:ss UT) *	Latitude of Su-solar Point, (degrees North) **	Time of Prime Meridian Alignment with Sub-solar Point (hh:mm:ss UT)	Latitude of Su-solar Point, (degrees North)	Difference, SEE - STK (seconds)
1 Jan 1980	12:04:48	-23	12:03:18	- 23.05	90
1 Jan 1990	12:04:36	-23	12:03:33	-23.00	63
1 Jan 2000	12:04:08	-23	12:03:18	-23.02	50
30 June 2000	12:04:29	23	12:03:43	23.13	46
1 Jan 2001	12:04:15	-23	12:03:41	-22.97	34
1 Jan 2010	12:03:48	-23	12:03:34	-22.99	14
1 Jan 2020	12:03:10	-23	12:03:21	-23.02	-11
1 Jan 2040	12:02:21	-23	12:03:22	-23.01	-61

* Estimated error in time of sub-solar point alignment with Prime Meridian was ± 30 seconds, or ± 0.125 degrees longitude (based on a sidereal day of 23h 56 m 4s).

** Estimation of the sub-solar latitude within the SEE could only be made to ± 1 degree.

1. Sub-solar alignment with the equatorial plane for the year 2000. The time of the year for which the sun was aligned with the true of date vernal equinox for the year 2000 was also estimated. This time occurs when the sun crosses the equatorial plane traveling in a northward direction.
    a. SEE
        i. Time of zero latitude for the sub-solar point was estimated to be 20 March 2000 17:00:00, at about 73 degrees **west** longitude.
        ii. Due to the slow motion of the sun in the north-south direction (caused by Earth motion about the Sun), there is a large time of occurrence error on this number. For ~1/3 of a line width movement of the sub-solar point (the approximate width of the texture map equator line center pixel), the error estimation in this time estimate is +/- 10 hours, or +/- 150 degrees longitude.
        iii. At the apparent time of the equatorial crossing for the SEE model (20 March 2000 17:00:00), the STK model indicated a sub-solar latitude of only 0.16 degrees, which could be used as a basis for the error in estimation of object latitude within the SEE.
    b. STK
        i. Time of zero latitude for the sub-solar point was extracted to be 20 March 2000 07:27:00.00, at a longitude of 70.11 degrees.
        ii. A +/- 10 hour window in the STK data indicates a north-south sub-solar point movement of only +/- 0.165 degrees.

5.3 Test Case 3:

Objective:

The goal of this series of tests was to estimate the accuracy of the position of the primary planets and the Earth Moon, relative to a well respected planetary ephemeris data source. The data source chosen for comparison was the JPL Horizons website, which reports out ephemeris data for solar system objects up to 2040 AD (depending on the object/body, ephemeris data may only be available through ~2025). The JPL ephemerides are based upon numerical integration techniques that calculate the position and velocity of objects due to multi-body perturbations.

Comparison was performed by extracting the position and velocity of each object of interest, relative to its primary central body, from the SEE and comparing the results to values obtained from the Horizons website. For each of the primary planets (Mercury through Pluto), this meant differencing the JPL and SEE position data relative to the center of the Sun. For the Earth Moon, this meant differencing the Moon position relative to the center of the Earth.

Four plots of the resulting data are presented for each planet or moon. In the first two plots, the absolute difference in position between the JPL and SEE data is shown in km, and as a percentage of the orbit semi-major axis. The second two plots show the difference in velocity between JPL and SEE for the object, both in km/s and as a percent of the average object speed.

Plots showing the position and velocity comparisons of the SEE with Horizons data are provided in Figures 22 through 37.

Earth

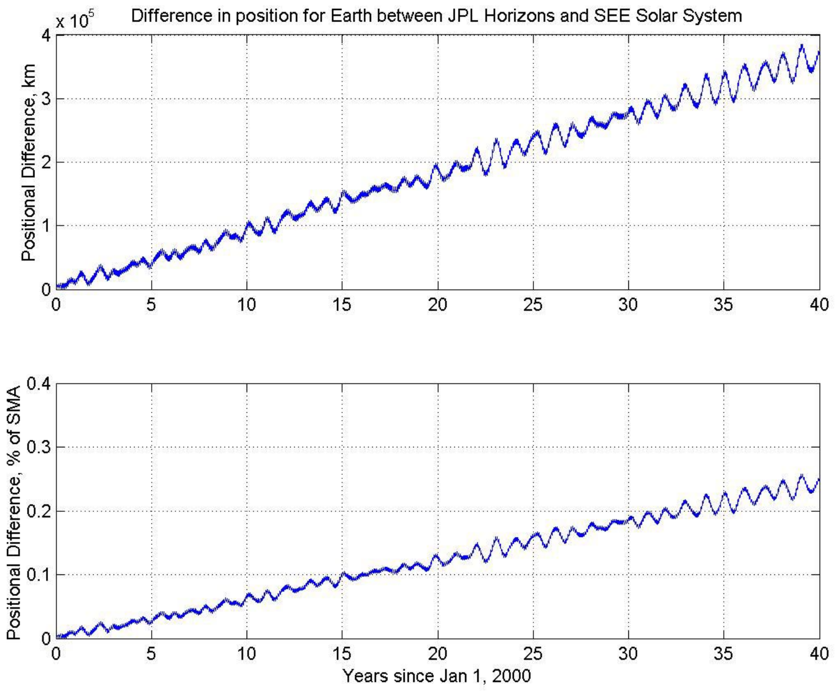


Figure 20. Comparison of Earth Position from JPL Horizons and SEE for the years 2000-2040

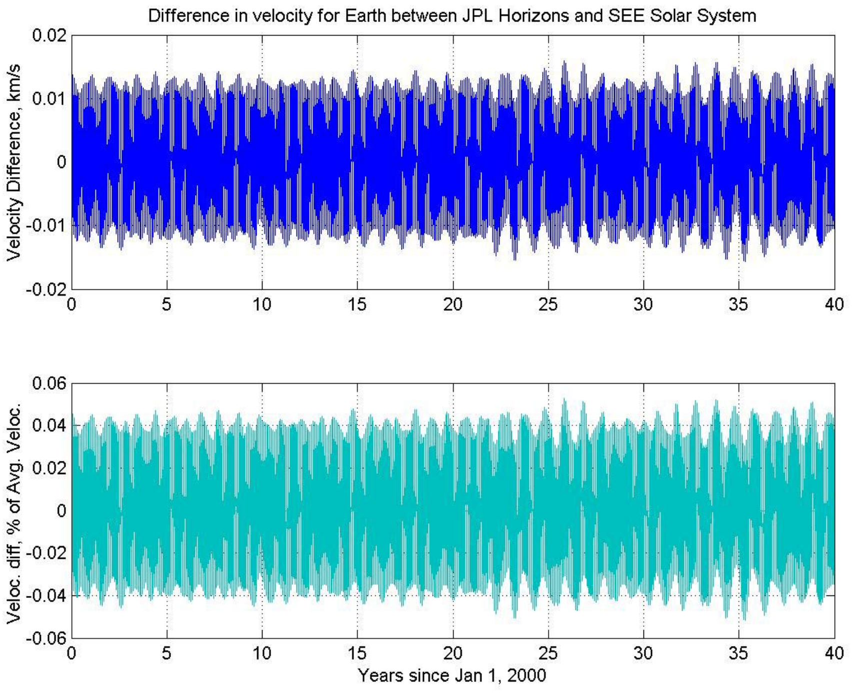


Figure 21. Comparison of Earth Velocity from JPL Horizons and SEE for the years 2000-2040

Earth Moon

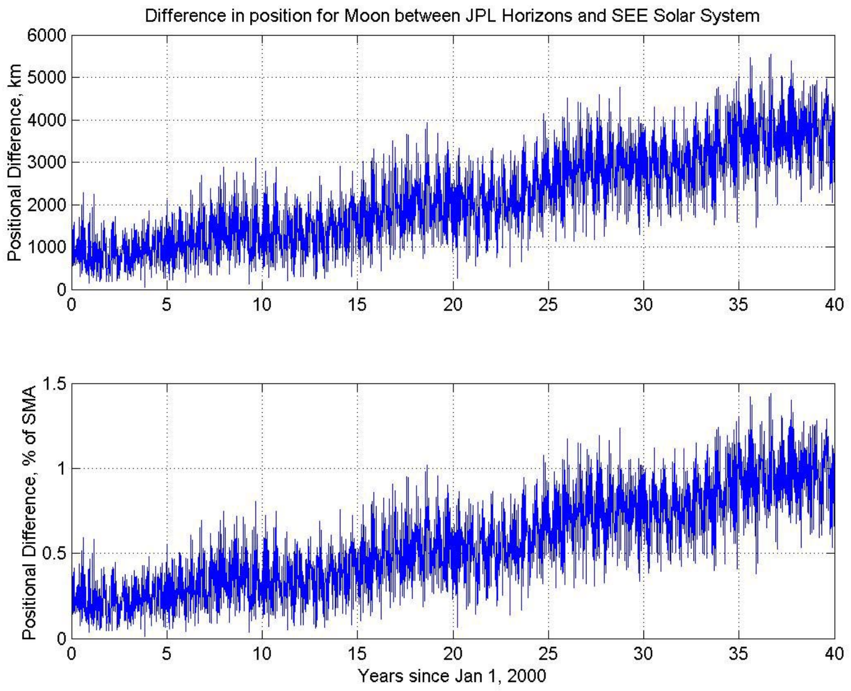


Figure 22: Comparison of Moon Position from JPL Horizons and SEE for the years 2000-2040

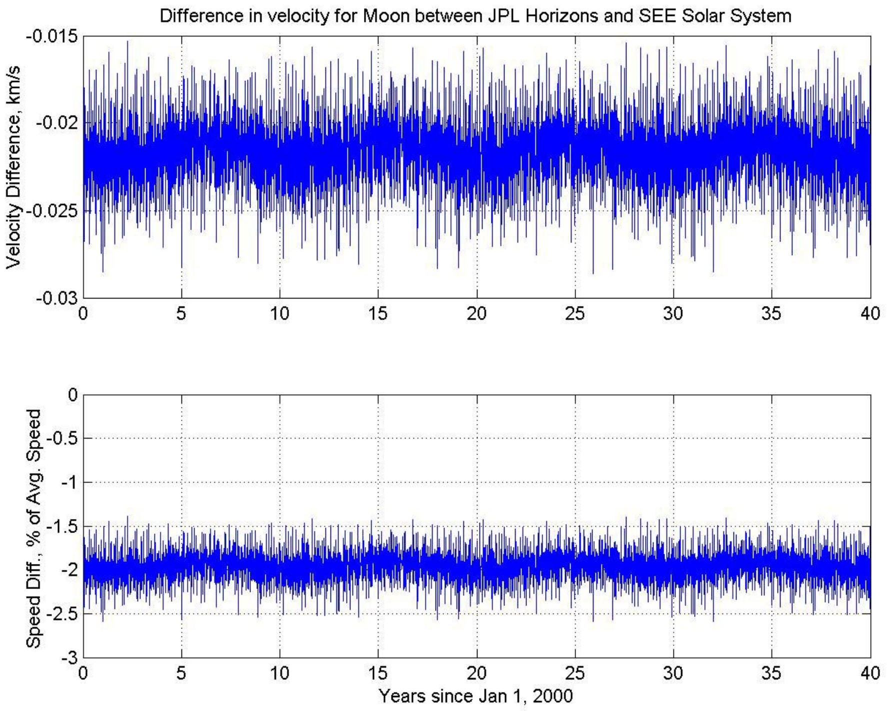


Figure 23: Comparison of Moon Velocity from JPL Horizons and SEE for the years 2000-2040

Mercury

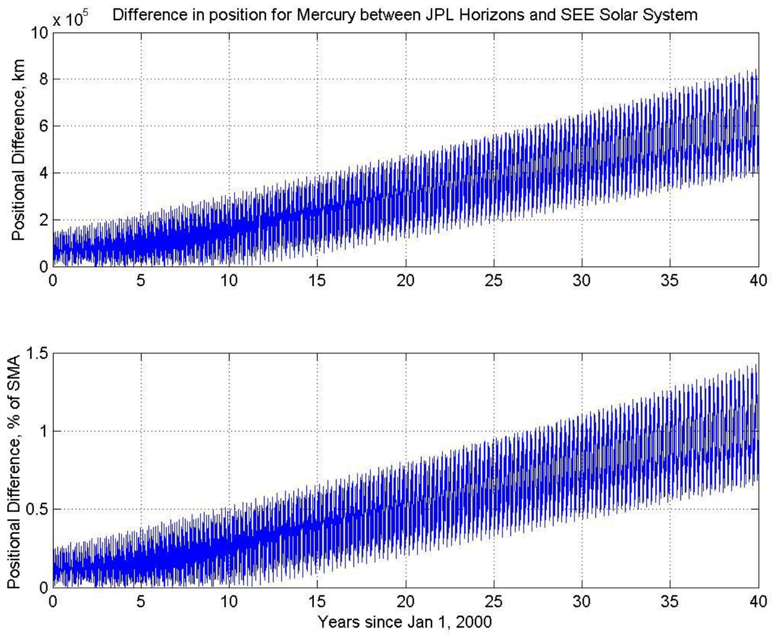


Figure 24: Comparison of Mercury Position from JPL Horizons and SEE for the years 2000-2040

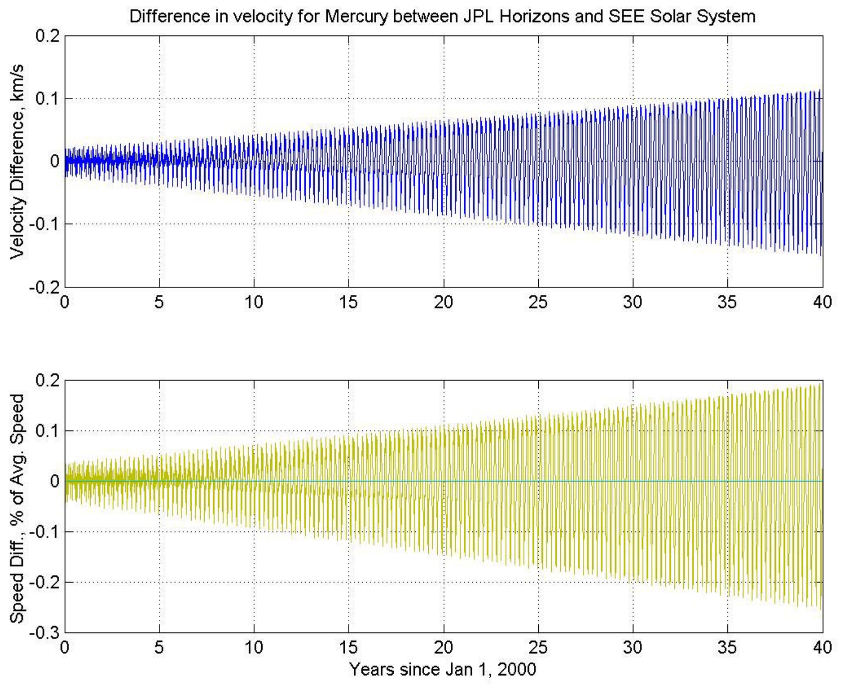


Figure 25: Comparison of Mercury Velocity from JPL Horizons and SEE for the years 2000-2040

Venus

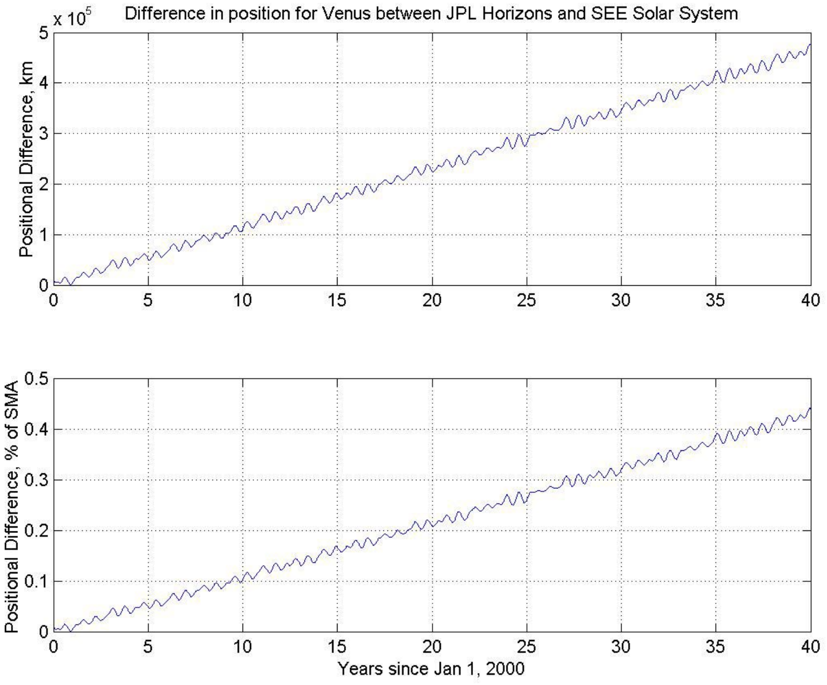


Figure 26: Comparison of Venus Position from JPL Horizons and SEE for the years 2000-2040

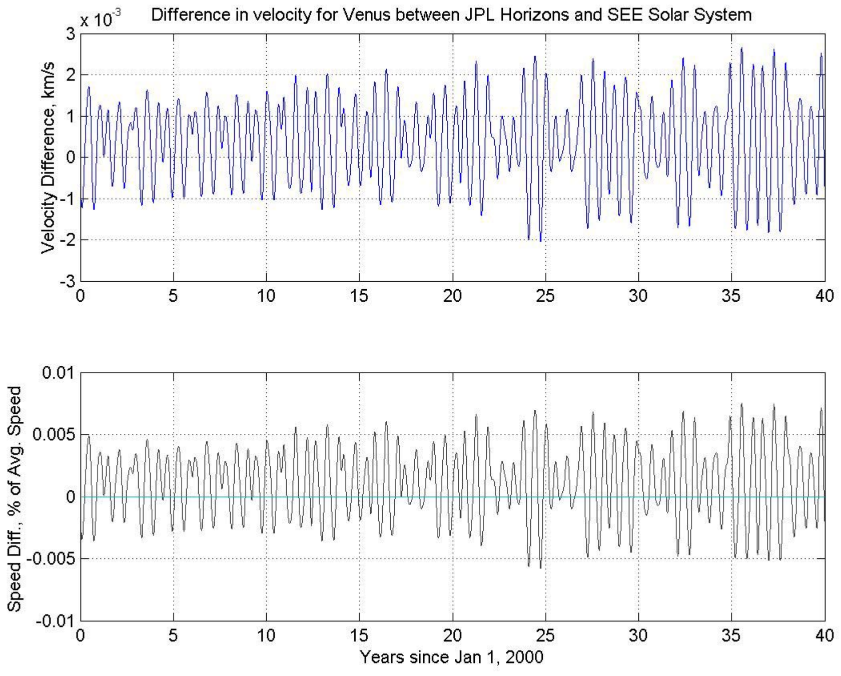


Figure 27: Comparison of Venus Velocity from JPL Horizons and SEE for the years 2000-2040

Mars

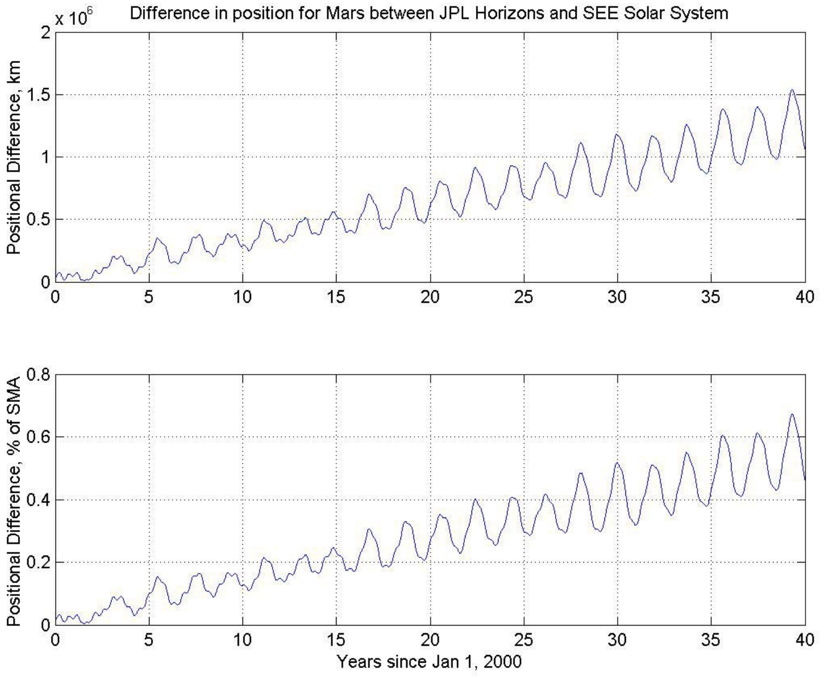


Figure 28: Comparison of Mars Position from JPL Horizons and SEE for the years 2000-2040

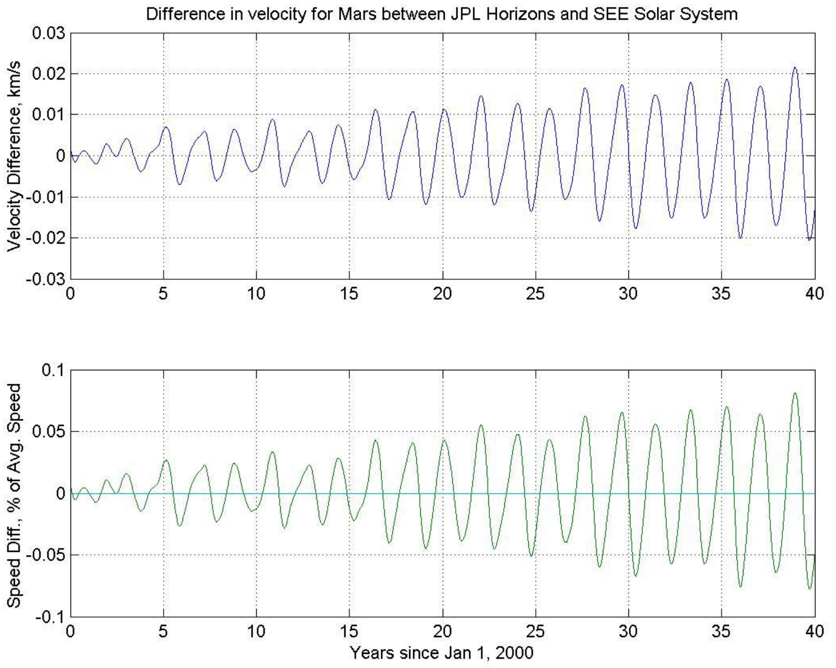


Figure 29: Comparison of Mars Velocity from JPL Horizons and SEE for the years 2000-2040

Jupiter

Note: JPL Horizons data not available for Jupiter past 31 December 2024 00:00:00.

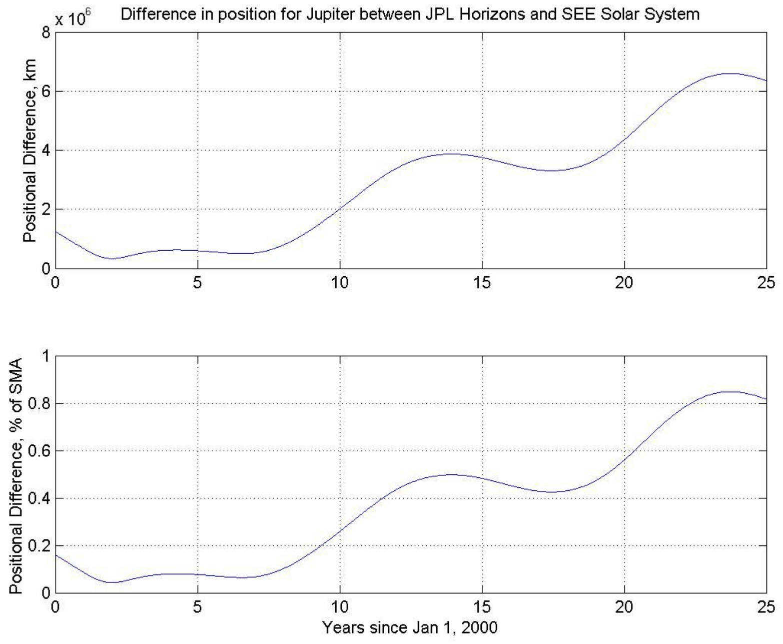


Figure 30: Comparison of Jupiter Position from JPL Horizons and SEE for the years 2000-2025

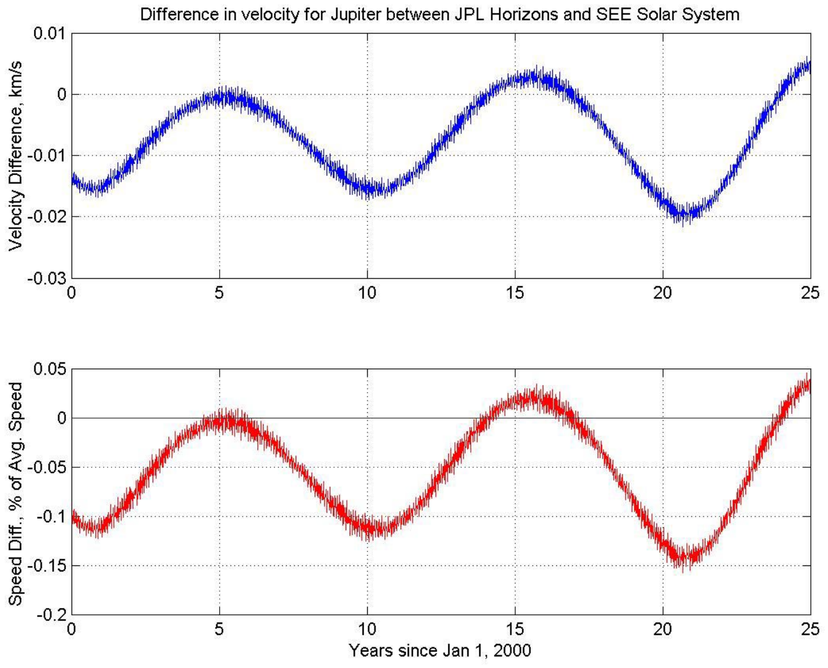


Figure 31: Comparison of Jupiter Velocity from JPL Horizons and SEE for the years 2000-2025

Saturn

Note: JPL Horizons data not available for Saturn past 16 January 2025 00:00.00.

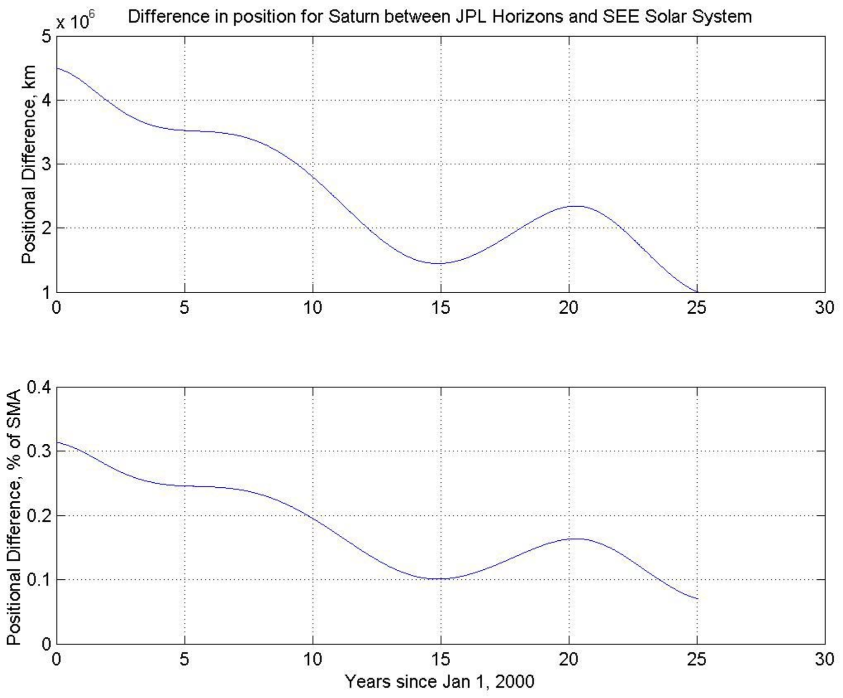


Figure 32: Comparison of Saturn Position from JPL Horizons and SEE for the years 2000-2025

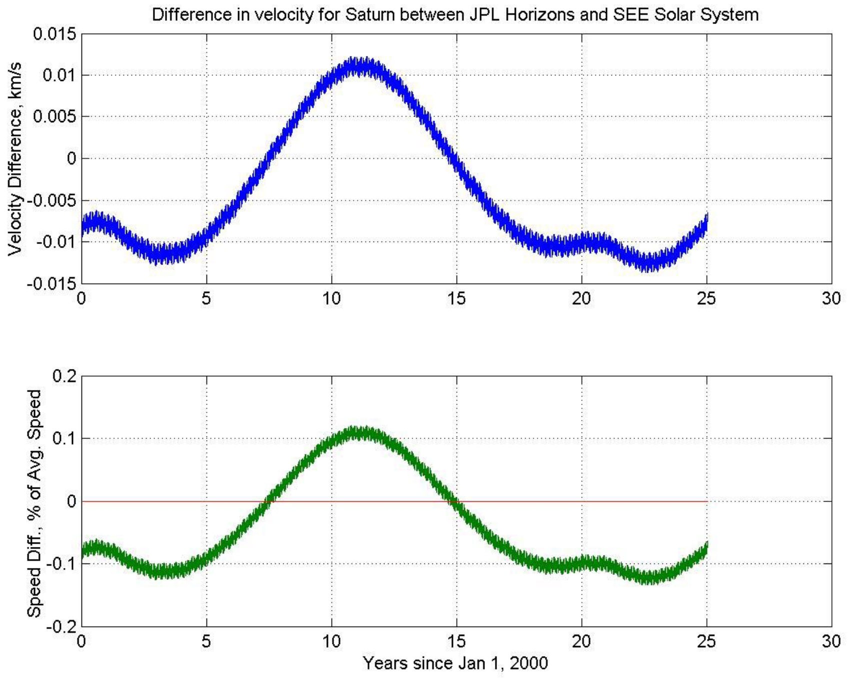


Figure 33: Comparison of Saturn Velocity from JPL Horizons and SEE for the years 2000-2025

Uranus

Note: JPL Horizons data not available for Uranus past 4 January 2025 00:00.00.

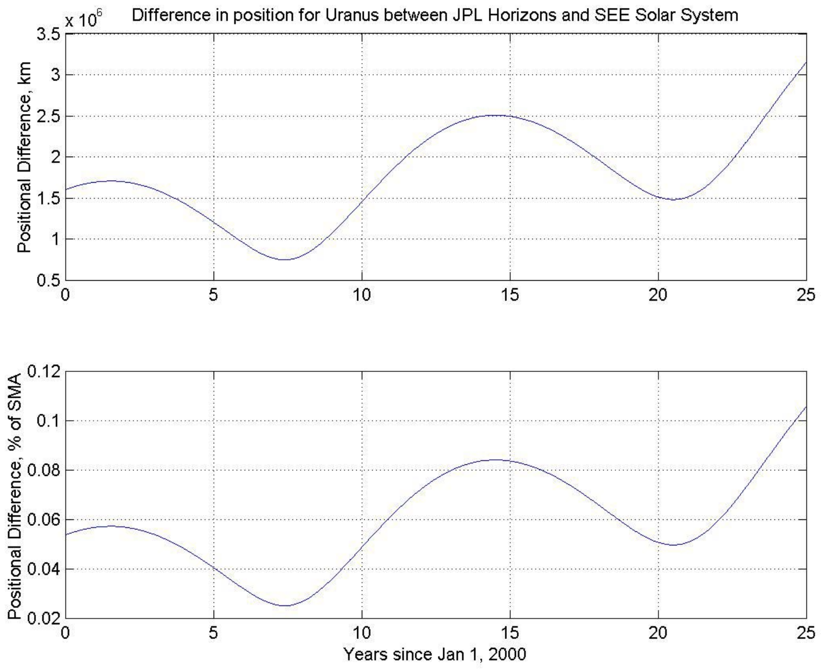


Figure 34: Comparison of Uranus Position from JPL Horizons and SEE for the years 2000-2025

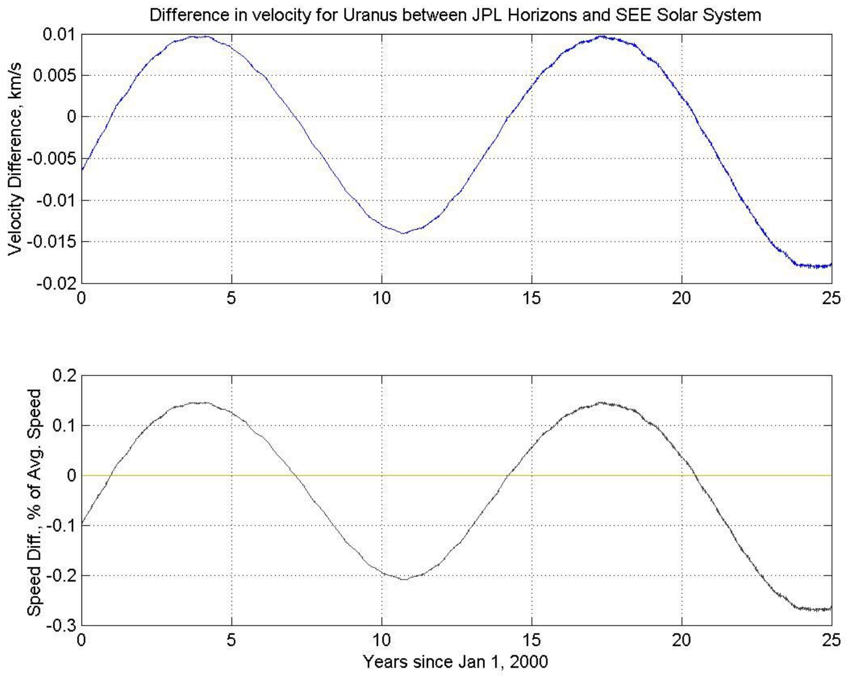


Figure 35: Comparison of Uranus Velocity from JPL Horizons and SEE for the years 2000-2025

Neptune

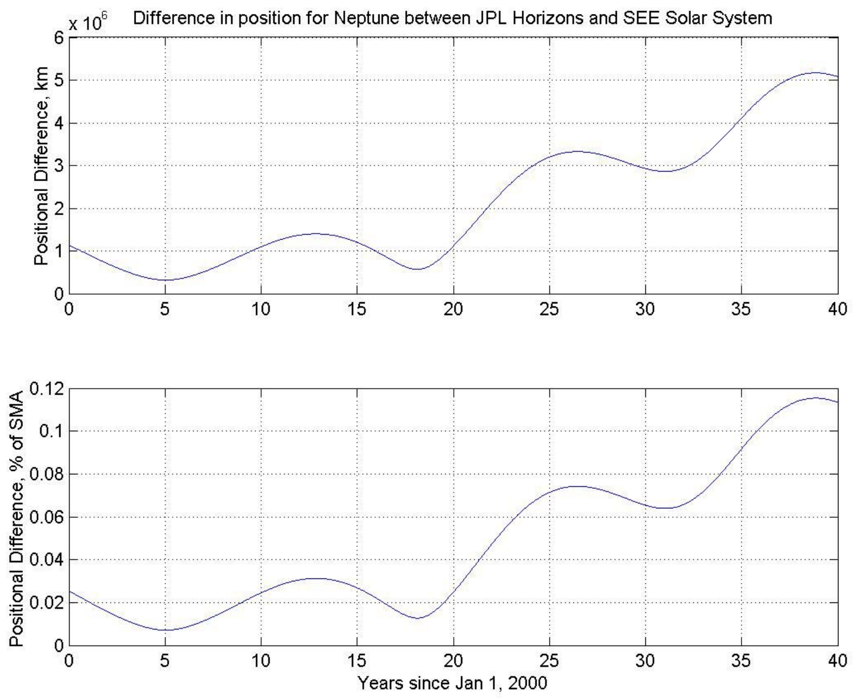


Figure 36: Comparison of Neptune Position from JPL Horizons and SEE for the years 2000-2040

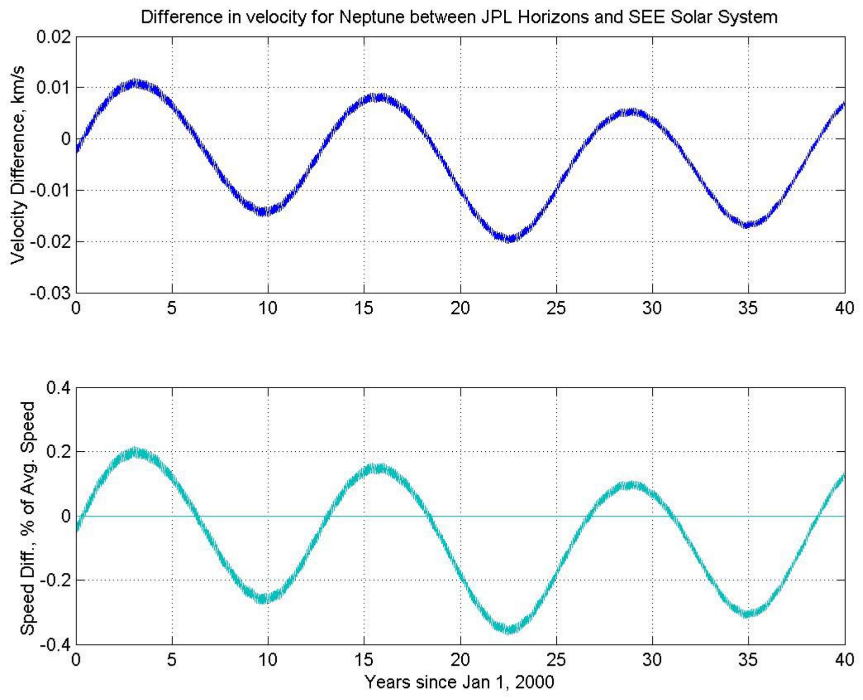


Figure 37: Comparison of Neptune Velocity from JPL Horizons and SEE for the years 2000-2040

Pluto

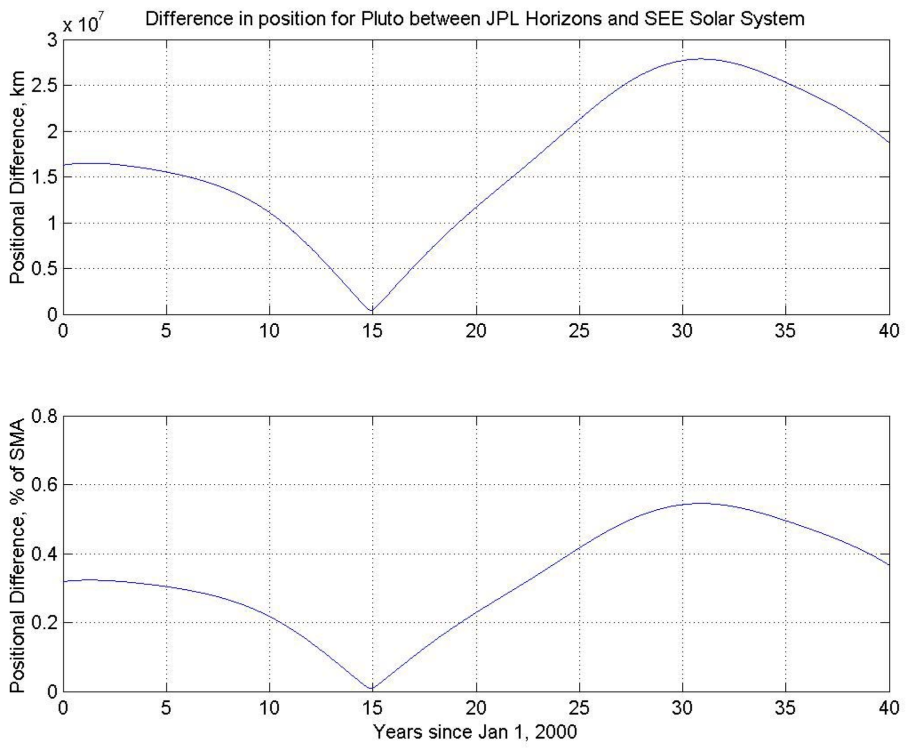


Figure 38: Comparison of Pluto Position from JPL Horizons and SEE for the years 2000-2040

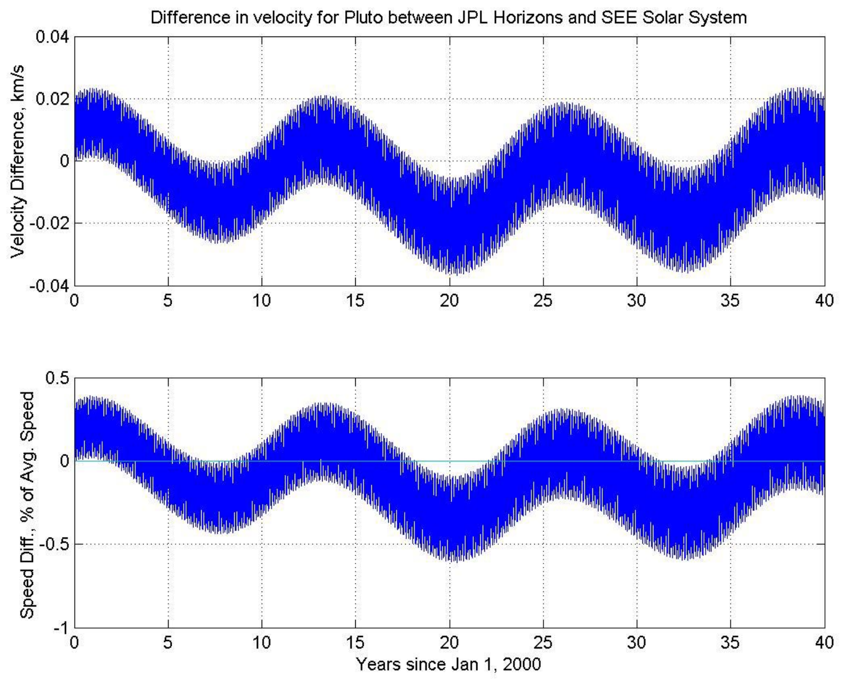


Figure 39: Comparison of Pluto Velocity from JPL Horizons and SEE for the years 2000-2040

Discussion:

Instructions for Obtaining JPL Horizons Ephemeris Data:

The process for obtaining the desired ephemeris data from the JPL Horizons system is fairly straightforward, and well described in the Horizons User's manual, located at 'http://ssd.jpl.nasa.gov/horizons_doc.html'. A brief version of the process is documented here to facilitate a quick update of the ephemeris data, if necessary.

Although there are three methods for accessing data in the Horizons database (telnet, WWW, and email), the easiest and most thorough (as far as amount of data available) was determined to be the telnet method, described below. The WWW interface has limitations that make it unusable for obtaining the data necessary for generation of the comparison plots in this report. The data obtained for these plots was the Cartesian coordinates of the center of each body relative to the Sun, and reported in the J2000 coordinate frame. At any point in the Horizons system help can be obtained by typing '?' (or '?!' for extended help).

To obtain different or updated Horizon's data:

1. Open a browser window, and go the Horizons Website URL: 'http://ssd.jpl.nasa.gov/horizons.html'.
2. Click on the telnet link at the top of the page. This should open a telnet window automatically within MS Windows. If this fails, manually open a telnet session with 'telnet ssd.jpl.nasa.gov 6775'. This should anonymously log into the Horizon's system, and result in a 'Horizons>' system prompt.
3. Enter the object number or name of the object for which position and velocity information is desired. Each object has a specific number assigned to it, for example, the planet Mercury is ID 199, while the Mercury barycenter is ID 1. A search is possible by using an asterisk (*) as a wildcard in the name of the planet desired.
4. Once the object of interest is selected, a summary of the object information (object gravitational constant, diameter, etc.) will be displayed. At this point, you can select to have the information emailed to you, select to retrieve it via FTP, or continue to the object ephemeris.
5. Selecting 'Ephemeris' by typing 'E' will bring you to a prompt which allows you to retrieve the observed position of the object from a location on the Earth's surface (observe,'o'), the planet's orbital elements (elements, 'e'), or positional vector (vectors, 'v'). Select 'v'.
6. The coordinate center for the positional information must be supplied. Type '500', which corresponds to the Sun's body center location. This same information could have been extracted from the Horizons system by typing '* @ sun', which would report all body locations on/in the sun that are part of the database. Type '?' for more help.
7. The next selection is to define the reference plane for the data. The three selections are 'frame', which is the Earth's mean equator and equinox of the

reference Epoch, 'body', which is the selected body's equator and node of date, or 'eclip', which is the mean ecliptic and equinox of the reference epoch. Select 'eclip'.

8. Next, the starting and ending dates for the desired data need to be entered in the format indicated.
9. The output interval for the desired data also needs to be specified. The comparison plots were done on an interval of 5 days, or '5d'.
10. The output table default values for coordinate frame, light travel time corrections, units of distance and time measure, output format, output type , and header labels are displayed. If the user desires to use the indicated quantities, then a 'y' or carriage return will accept the values and continue to the next step.
    a. Make sure that for these comparisons that the Ref. Frame = ICRF/J2000, Corrections = NONE, Units = km-s, CSV format = yes, table format = 03, and vector label = yes.
11. The requested data is now generated and displayed one screen at a time. When done inspecting the data to make sure that it is the information desired, type 'q' to skip the rest of the display.
12. The data just displayed can now be retrieved through FTP, Kermit, or Mail (email). Method of retrieval is up to the user, and directions for retrieval are given for each method selected.
13. Selecting 'Again' or 'A' will keep the same body of interest and allow the user to select more information for retrieval (such as orbital elements). It is equivalent to starting the above process at step 5.
14. Selecting 'New Case' or 'N' will put the user back at the original Horizon's prompt, allowing them to select a new body of interest.

6 ABBREVIATIONS AND ACRONYMS

API	Application Programming Interface
CAPS	Comet and Asteroid Protection System
DAC	Design Analysis Cycle
DDEX	Dynamics Data Extractor
ISS	International Space Station
JSC	Johnson Space Center
LaRC	Langley Research Center
RASC	Revolutionary Aerospace Systems Concepts
SEE	Synergistic Engineering Environment
SEMDA	Systems Engineering, Modeling and Data Analysis
STL	Stereolithography
VIPER	Vehicle Integrated Performance and Resources
VRML	Virtual Reality Modeling Language

7 NOTES

1. The SEE public home page http://centauri.larc.nasa.gov/see contains examples of visualizations created with the SEE software. These include the evaluation of station contingency maneuvers planned for the event of a dual failure of the primary and redundant Plasma Contactor Units, a review of the station flight attitude using the playback of actual ISS telemetry streams, and an analysis of the data-taking opportunities of the SAGE III experiment (an externally mounted payload that requires a line-of-sight to the Sun at periods near dusk and dawn.)

2. The distinction made here between *internal* and *external* programs is based on whether the code is compiled into the SEE software itself, or runs as a separate executable. Examples of external programs used for Build I are the Space Station Rigid Body Dynamics Simulator (SSRMBS) developed at JSC, and DOCKSIM, a vehicle docking simulation developed at Langley Research Center.

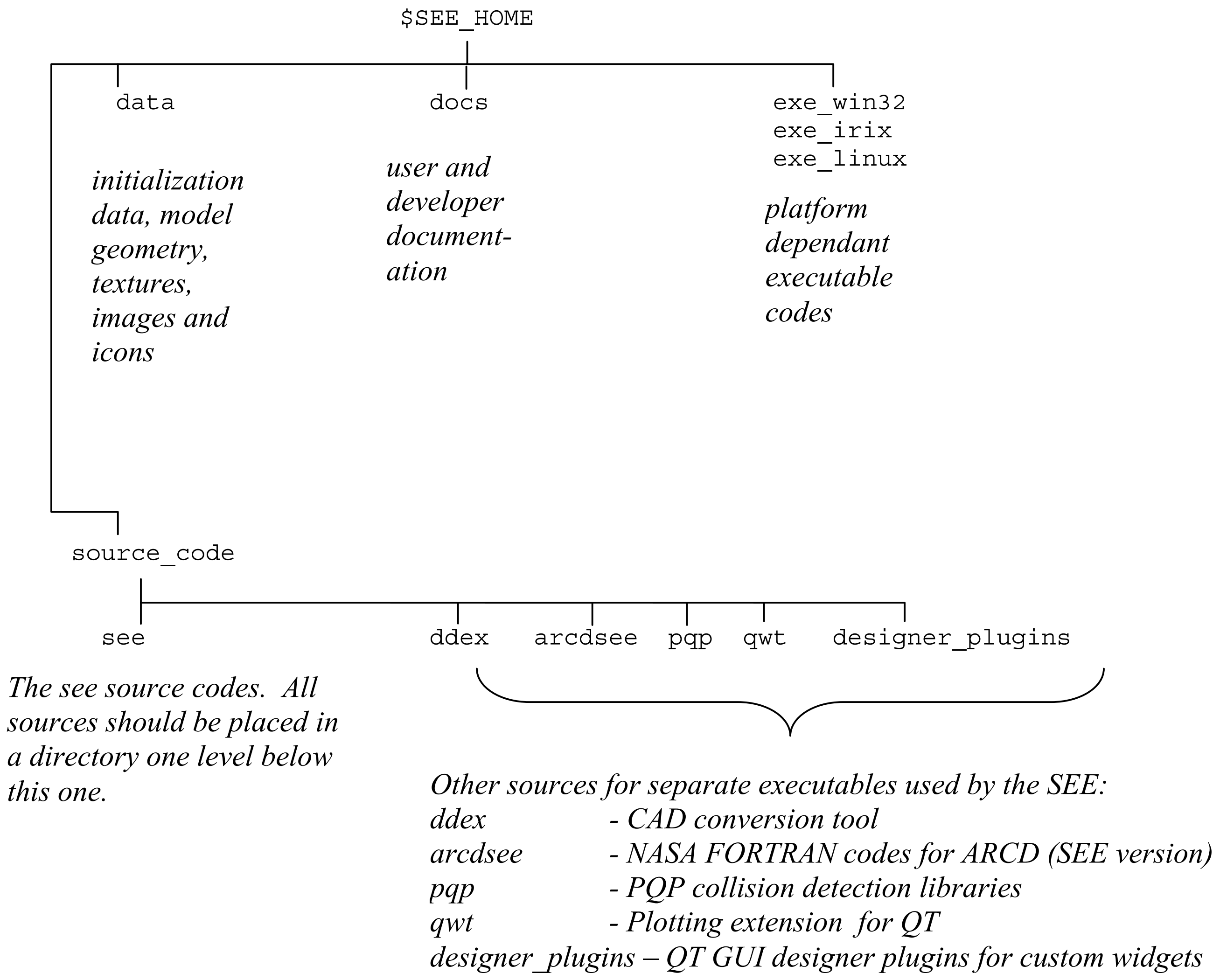
3. In Build I, shared memory and TCP/IP packets were used to communicate signals from the GUI to the main SEE process.

4. Since the QT timer is already allowing for the processing of user inputs before allowing the event loop to proceed , an explicit *respond* phase may prove to be unnecessary.

5. The joint location information needed by the visualization application to properly place the station rigid bodies is not present in the SEMDA model file. The I-DEAS application is used to add this data after the models are received from the SEMDA lab but before the model file is exported to an ISS SEE compatible format by the DDEX program.

8 APPENDIX A: SEE DIRECTORY STRUCTURE

READ ONLY DIRECTORIES:



initialization data, model geometry, textures, images and icons

user and developer document-ation

platform dependant executable codes

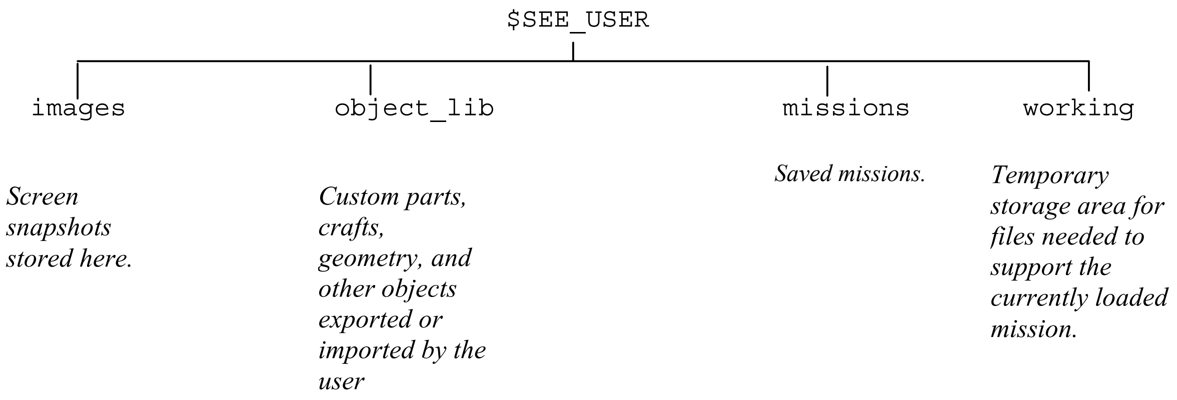
The see source codes. All sources should be placed in a directory one level below this one.

Other sources for separate executables used by the SEE:

ddex	*- CAD conversion tool*
arcdsee	*- NASA FORTRAN codes for ARCD (SEE version)*
pqp	*- PQP collision detection libraries*
qwt	*- Plotting extension for QT*
designer_plugins	*– QT GUI designer plugins for custom widgets*

Appendix A: SEE Directory Structure (continued)

USER (write-able) DIRECTORIES:



Screen snapshots stored here.

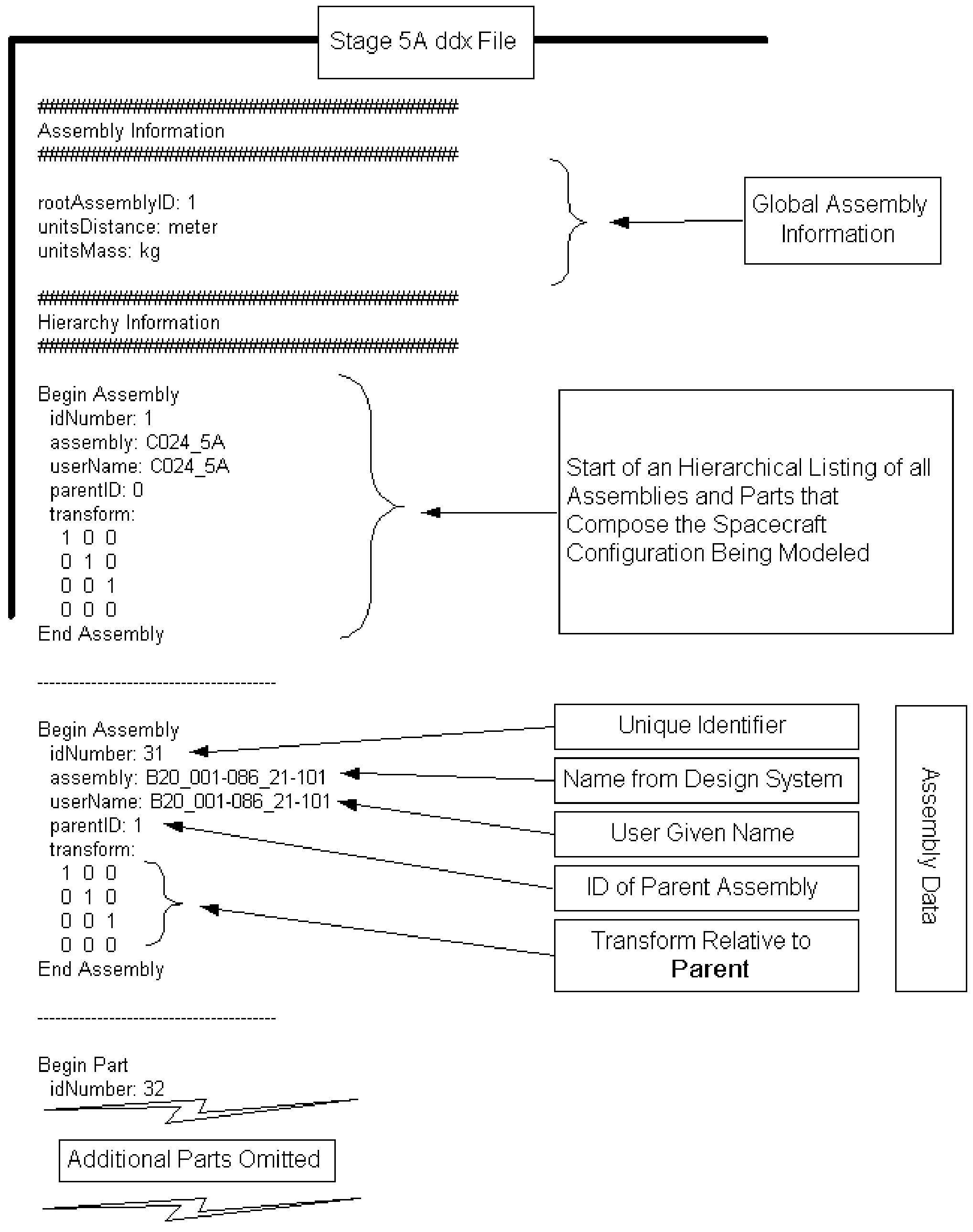
Custom parts, crafts, geometry, and other objects exported or imported by the user

Saved missions.

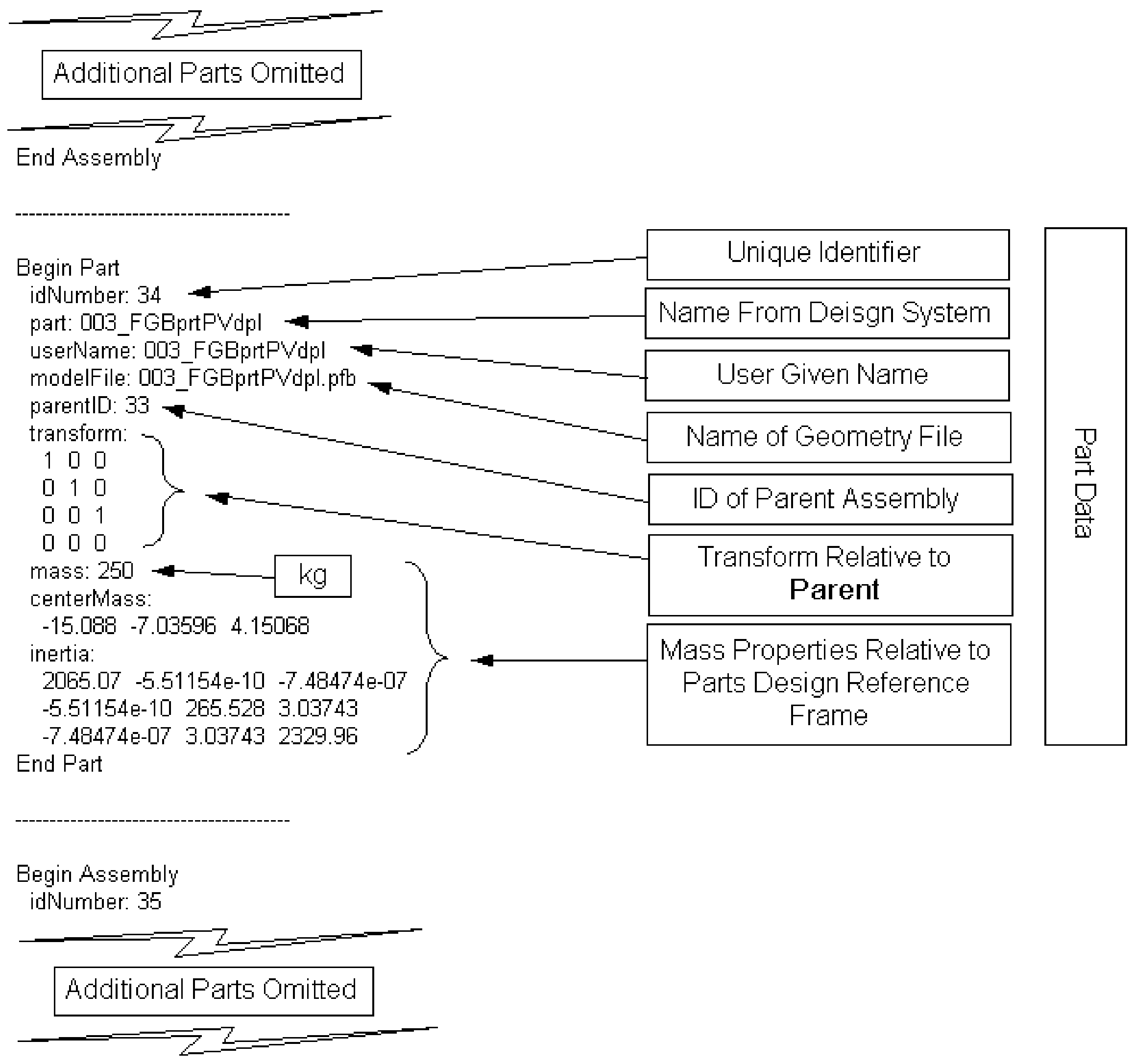
Temporary storage area for files needed to support the currently loaded mission.

9 APPENDIX B: ANNOTED DDEX FILE

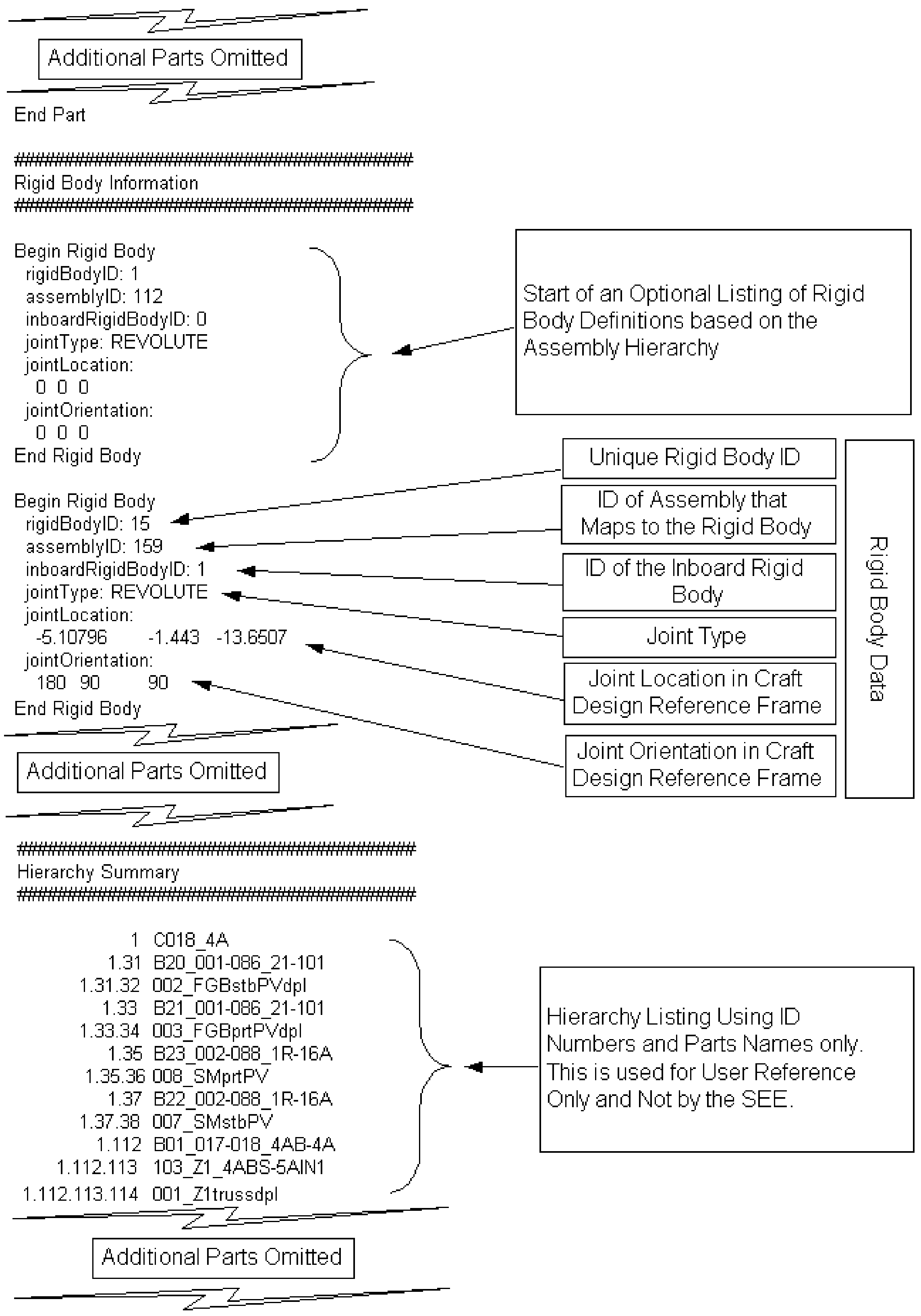
This annotated example of a configuration (.ddex file) corresponds to Stage 5A of the DAC 8 assembly sequence.


Stage 5A ddx File
Global Assembly Information
Start of an Hierarchical Listing of all Assemblies and Parts that Compose the Spacecraft Configuration Being Modeled
Unique Identifier
Name from Design System
User Given Name
ID of Parent Assembly
Transform Relative to **Parent**
Assembly Data
Additional Parts Omitted

```
##################################################
Assembly Information
##################################################

rootAssemblyID: 1
unitsDistance: meter
unitsMass: kg

##################################################
Hierarchy Information
##################################################

Begin Assembly
  idNumber: 1
  assembly: C024_5A
  userName: C024_5A
  parentID: 0
  transform:
    1  0  0
    0  1  0
    0  0  1
    0  0  0
End Assembly

----------------------------------------

Begin Assembly
  idNumber: 31
  assembly: B20_001-086_21-101
  userName: B20_001-086_21-101
  parentID: 1
  transform:
    1  0  0
    0  1  0
    0  0  1
    0  0  0
End Assembly

----------------------------------------

Begin Part
  idNumber: 32
```



```
Additional Parts Omitted

End Assembly

----------------------------------------

Begin Part
  idNumber: 34
  part: 003_FGBprtPVdpl
  userName: 003_FGBprtPVdpl
  modelFile: 003_FGBprtPVdpl.pfb
  parentID: 33
  transform:
    1  0  0
    0  1  0
    0  0  1
    0  0  0
  mass: 250
  centerMass:
    -15.088  -7.03596  4.15068
  inertia:
    2065.07  -5.51154e-10  -7.48474e-07
    -5.51154e-10  265.528  3.03743
    -7.48474e-07  3.03743  2329.96
End Part

----------------------------------------

Begin Assembly
  idNumber: 35

Additional Parts Omitted
```



```
Additional Parts Omitted

End Part

#############################################
Rigid Body Information
#############################################

Begin Rigid Body
  rigidBodyID: 1
  assemblyID: 112
  inboardRigidBodyID: 0
  jointType: REVOLUTE
  jointLocation:
    0  0  0
  jointOrientation:
    0  0  0
End Rigid Body

Begin Rigid Body
  rigidBodyID: 15
  assemblyID: 159
  inboardRigidBodyID: 1
  jointType: REVOLUTE
  jointLocation:
   -5.10796       -1.443   -13.6507
  jointOrientation:
    180   90        90
End Rigid Body

Additional Parts Omitted

#############################################
Hierarchy Summary
#############################################

             1   C018_4A
          1.31   B20_001-086_21-101
       1.31.32   002_FGBstbPVdpl
          1.33   B21_001-086_21-101
       1.33.34   003_FGBprtPVdpl
          1.35   B23_002-088_1R-16A
       1.35.36   008_SMprtPV
          1.37   B22_002-088_1R-16A
       1.37.38   007_SMstbPV
         1.112   B01_017-018_4AB-4A
     1.112.113   103_Z1_4ABS-5AIN1
 1.112.113.114   001_Z1trussdpl

Additional Parts Omitted
```

Start of an Optional Listing of Rigid Body Definitions based on the Assembly Hierarchy
Unique Rigid Body ID
ID of Assembly that Maps to the Rigid Body
ID of the Inboard Rigid Body
Joint Type
Joint Location in Craft Design Reference Frame
Joint Orientation in Craft Design Reference Frame
Rigid Body Data
Hierarchy Listing Using ID Numbers and Parts Names only. This is used for User Reference Only and Not by the SEE.